NORTH-HOLLAND

OVERVIEW OF DASWAM: EXPLOITATION OF DEPENDENT AND-PARALLELISM*

KISH SHEN

▷ The Dynamic Dependent And-parallel Scheme (DDAS) is a parallel execution scheme for Prolog that is designed to exploit independent and dependent and-parallelism in full Prolog programs. In this paper, an overview of some techniques for implementing DDAS is presented. The main purpose of these techniques is to provide reasonably efficient methods for implementing the idea of the dependent status and the dynamic detection of the leftmost instance of a variable, along with the techniques needed for suspending and waking up of goals. DASWAM, an implementation of DDAS using the techniques presented in this paper, is based on modifying the Warren Abstract Machine (WAM) for parallel execution, and is designed to execute programs efficiently. A prototype DASWAM has been implemented, and results on running significant application programs presented in this paper suggest that it is possible to implement DASWAM efficiently, and that significant parallelism can be extracted. ◁

1. INTRODUCTION

Logic programming languages provide a separation of logic and control [14], and thus allow much freedom in how a program can be executed (the "control").[1] One possible way to increase the execution speed of logic programming languages

*Most of the research reported in this paper was carried out while the author was at the Computer Laboratory, University of Cambridge, Cambridge, U.K., and the Department of Computer Science, University of Bristol, Bristol, U.K.

[1]In practice, this freedom is constrained by the "impure" features of a practical logic programming language, and also by other practical considerations.

Address correspondence to Kish Shen, Centre for Novel Computing, Department of Computer Science, University of Manchester, Manchester M13 9PL, United Kingdom.

Received June 13, 1996.

655 Avenue of the Americas, New York, NY 10010

0743-1066/96/$15.00
PII S0743-1066(96)00079-9

is through parallelism, where the flexibility of control allows parallelism to be exploited implicitly.

Implicit exploitation of parallelism in logic programming is attractive because it retains all the advantages of the logic programming approach, allowing an increase in execution speed, but without an increase in programming complexity. Many implicit parallel logic programming systems have been proposed, including a few that have been developed and approach sequential Prolog in stability and usability (e.g., [1, 15]).

Logic programming languages provide many opportunities for parallel execution. The parallelism can be classified most generally into two types:

Or-parallelism: The execution of a logic program can be regarded as a process of finding proof(s). In general, there may be many paths to finding the proofs. Or-parallelism arises if potentially more than one of these paths are explored in parallel.

And-parallelism: Each path of an attempt at a proof may contain subproofs. And-parallelism arises when these subproofs can potentially be tried in parallel. If the execution of the subproofs is known to *not* affect the execution of other subproofs that are executed in parallel, then the parallelism is known as **independent and-parallelism (IAP)**. If the execution might affect others, then the parallelism is known as **dependent and-parallelism (DAP)**.[2]

Currently, and-parallelism has more basic open research issues than or-parallelism. For example, it is comparatively easy to extract or-parallelism from logic programs, as the parallelism arises from executing the alternatives in parallel. For and-parallelism, however, some execution model has to be defined so that the proof can be split into subproofs which are convenient for execution in parallel. As a proof can be split many ways, and the different ways of splitting it can have profound consequences on the amount of extracted parallelism and computational complexity, the and-parallel characteristics are very dependent on the model defined to extract them.

This paper presents DASWAM (Dynamic And-parallel SICStus WAM), an implementation scheme for DDAS [26, 27], a parallel execution model for full Prolog that exploits DAP, with IAP as a special subcase. Prolog is used as the base language mainly because it provides a familiar programming environment, and allows existing Prolog programs to be executed under the model, with possible increases in execution speed due to parallelism. Extensions to Prolog, both those to take advantage of the opportunities that parallel execution offers and those that improve on Prolog itself (such as constraint handling), can also be added to the execution model.

A prototype implementation of DASWAM which simulates parallel execution by "time-slicing" at the abstract machine level has been developed, and is currently able to execute real applications programs in and-parallel. This approach was chosen for several reasons: first, it allows a much more detailed examination of various characteristics of the system without fear of perturbing its behavior; second, it allows much simpler debugging and thus development of the system; third, it allows

[2]Note that under this definition, some forms of parallelism that have sometimes been classified separately from and-parallelism are considered to be forms of and-parallelism, such as unification parallelism and recursion parallelism.

the prototype to be very portable. Much of the behavior of a parallel DASWAM is accurately duplicated, though one area that might not be adequately simulated is the speedup, as speedups of a parallel implementation are affected by many complex factors; the "speedups" provided by the prototype in executing a program can best be viewed as the potential parallelism of the program. Thus, valuable insights into DDAS are provided as well: the results can be used to see if DDAS is indeed able to extract parallelism from programs, and also to allow an eventual parallel implementation's speedups to be placed in context.[3]

It is beyond the scope of this paper to discuss every aspect of DASWAM. One aim of this paper is to provide an overview of DASWAM, concentrating on some of the more novel aspects. The level of detail in the presentation is intended to convey the main novel ideas needed for the implementation, but is certainly not designed to be a specification of the implementation: such a specification would be of little general interest and would also make the paper much too long. The DASWAM overview section of the paper is a revised presentation of some of the material in [28, 29].

A second aim of the paper is to provide a report on the current (mid-1995) state of DASWAM by presenting and analyzing results for DASWAM running realistic application programs. In previous publications [27, 28], results were provided mainly for smallish benchmark-type programs, whose purpose was mainly to verify that DDAS can indeed be implemented. The results presented in this paper are intended to be a more in-depth study of DASWAM and DDAS.

As this paper is designed to be relatively self-contained, the DDAS execution model shall also be presented. This is mainly a somewhat condensed and updated version of some of the material in [26, 29]. Some of the more subtle details are, however, not discussed, and the reader is referred to those papers and [27] for details.

This paper does not assume a familiarity with DDAS, but it does assume the reader has a reasonable knowledge of Prolog and its implementation, particularly the WAM [37]. It also assumes the reader has some familiarity with parallel implementation issues.

2. EXPLOITING DEPENDENT AND-PARALLELISM IN PROLOG: DDAS

2.1. Basic Approach

In common with many previous approaches to exploiting parallelism in logic programming languages (e.g., [7, 11, 15, 38]), DDAS takes the subtree-based approach to parallelism, i.e., parallelism is achieved by allowing more than one worker[4] to explore the search-space of a program simultaneously. Each worker performs work in much the same way as a sequential Prolog engine; thus the search-space is divided into "chunks" or subtrees that are each executed sequentially. Each such subtree is referred to as a **task**, and represents the execution of one or more goals—thus, the

[3]Very recently (after the initial writing of this paper), the prototype implementation was converted to execute programs in parallel. The conversion was carried out relatively easily, demonstrating the advantage of initially developing the prototype sequentially. The converted parallel system was quite stable from the start, and was able to run quite complex DAP and IAP programs with less than two man-months of work.

[4]worker is used to denote the entity which performs work (computation).

and-parallelism exploited by DDAS is at the goal level. A task finishes successfully if the goal(s) returns a solution.

In the subtree-based approach, equivalence to Prolog can be achieved by ensuring that the search-space explored in parallel is the same as that explored sequentially by Prolog,[5] i.e., corresponding unifications in the parallel and sequential executions make the same substitutions for variables to nonvariables, up to the renaming of variables. For or-parallelism and IAP, this equivalence is achieved because each task that executes in parallel cannot affect the computations performed in the other tasks, and each such task performs the same computation as in the equivalent part of the sequential search-space: thus the control can be viewed as a means of synchronizing the tasks such that when they execute, they are independent of other tasks which may execute in parallel. For DAP, the tasks running in parallel may affect each other's search-space, because there may be dependencies between such tasks. In order to maintain the equivalence to Prolog, DDAS synchronizes at the finer level of unifications instead of tasks. This synchronization ensures that the same computations are performed in parallel as in sequential Prolog.

DDAS can be viewed as consisting of two main components: the mechanism for synchronization, which ensures the equivalence to sequential Prolog during forward (normal) execution; and a backward execution component for ensuring that the equivalence is preserved when an unification fails. These components will now be presented in more detail.

2.2. Forward Execution

2.2.1. Basic Concepts and Definitions. A major goal of DDAS is to exploit DAP implicitly, that is, equivalence to sequential Prolog is maintained. The main problem with maintaining this equivalence is that, when tasks (working on different subtrees) are executed in parallel, they can potentially affect each other's execution, because dependencies between tasks are allowed in DDAS. Two tasks can be dependent on each other *if* they share unbound variables, and the important point is that they can affect each other's executions *only* through such variables.[6] In essence, the main idea behind DDAS is identifying the actions which cause these dependencies between tasks, and synchronizing these actions so that equivalence to Prolog is maintained.

Shared unbound variables can affect tasks executing in parallel if these tasks try to perform the following actions:

1. bind a shared unbound variable to a nonvariable term; this changes the state of the variable. As this variable is shared with other tasks, this change of state might affect the unifications performed in such tasks.
2. check the state of a shared unbound variable via metalogical predicates; as the state of such variables can be changed by another task.

It is important to note that performing such actions on a variable on one task does not automatically mean that it has an effect on other tasks that share the

[5]This is a simplified view that ignores differences due to speculative work and work avoided by intelligent backtracking.

[6]This ignores impure features of Prolog, which will be discussed in Section 6.

variable: it will only have an effect if these other tasks also try to perform these actions on the same variable. Therefore, only a subset of shared unbound variables will cause dependencies, and a more precise definition of such variables is needed.

Unbound variables which are shared by more than one task executing in parallel are known as *real dependent variables* if actions that causes dependency (as discussed above) can be performed on the variable by more than one task; otherwise the variable is *real nondependent*. A (nonground) term which contains one or more real dependent variables is known as a *real dependent term*; otherwise the term is a *real nondependent term*. Thus, real dependent variables/real dependent terms are precisely those variables/terms which would cause dependencies between tasks.

A binding x/t, where x is a real nondependent variable, is a *rv-binding* if t is a variable (either real dependent or nondependent). Otherwise (that is, if x is a real dependent variable, or t is a nonvariable), it is a *rnv-binding*.[7] For example, binding variable `V` (either real dependent or not) to the atom `foo` is a rnv-binding; binding variable `V` to variable `U` is a rv-binding unless both variables are real dependent variables.

A *rdep-access* is any access to a real dependent variable that is not a rv-binding, i.e., it is either a rnv-binding, or the checking of the state of an (unbound) dependent variable. All other accesses to variables/terms are nrdep-accesses.

The rdep-accesses are exactly the actions performed by a task that can affect other tasks executing in and-parallel. It precisely encompasses the two cases listed earlier where a task can affect other tasks executing in and-parallel: case 1 corresponds to a rnv-binding and case 2 corresponds to checking the state of a real dependent variable. The main idea behind DDAS is to synchronize these rdep-accesses so that equivalent behavior to Prolog is maintained: unifications which perform rdep-accesses are prioritized—unifications which would be executed first in sequential Prolog have a higher priority than those that would be executed later sequentially. That is, the left-to-right ordering of goals (and hence unifications and any accesses to variables they perform) of sequential Prolog is maintained for the priority system. Thus, for several tasks that share the same real dependent variable, the task that is executing the goal with the highest priority (the leftmost) is the task that is allowed to perform any rdep-accesses on the variable. This task is referred to as the *producer* of the variable. The other tasks must suspend if they try to rdep-access the variable, and are referred to as the *consumers* of the variable. Note, however, that they can still perform non-rdep-accesses to the variable, as these do not affect the computation of the consumer tasks.

A suspended task can be unsuspended and its execution allowed to resume when the action that originally caused the suspension can no longer affect the equivalence to Prolog behavior. The most obvious way for this to happen is if the action can no longer affect other tasks, i.e., if the action is no longer a rdep-access. This would occur if the real dependent variable that was being rdep-accessed becomes bound to a nonvariable by the producer of the variable. The consumer will then be accessing a bound value, instead of the original unbound real dependent variable, so the access is no longer a rdep-access.

[7] rnv- and rv-bindings are extensions of nv- and v-bindings, respectively, originally defined in [13].

However, this mechanism for unsuspending a task does not cover all possibilities: it is possible that the action that caused the suspension will *not* become a non-rdep-access. Consider the following program fragment:

```
foo(X) :- a(X), b(X).

a(_).
b(1).
```

assume that `a(X)` and `b(X)` are to be executed in and-parallel, sharing an unbound real dependent variable `X`: `a(X)` would be the producer for `X`, and `b(X)` the consumer. However, `a(X)` does *not* bind `X` in this example, and so if `b(X)` tries to rnv-bind `X` and suspends, the mechanism discussed so far is unable to unsuspend `b(X)`, causing a deadlock and a departure from the behavior of sequential Prolog.

To cope with this problem, another unsuspension mechanism is needed. Consider how sequential Prolog would execute the example: first `a(X)` would execute, without binding `X`, and then `b(X)` would execute when `a(X)` completes its execution; this implies that `b(X)`, which was initially in the consumer position, is allowed to rnv-bind the real dependent variable, because `a(X)`, the initial producer, does not rnv-bind the variable. Generalizing, to maintain equivalence to Prolog, consider the general case of executing the goals $G_1 \cdots G_n$ in a body in and-parallel; then the goal that is allowed to rdep-access a dependent variable is G_m, $1 \leq m \leq n$, iff the goals to the left of it do not rnv-bind the variable. However, the difficulty is that, in general, one is unable to determine (without executing the goals) that these goals will not rnv-bind the variable. Thus, when G_m tries to rdep-access the dependent variable, one cannot in general know that this rnv-binding can be performed, and the task has thus to be suspended. To unsuspend such tasks, DDAS introduces the concept of *dynamic producer*:

The producer for a real dependent variable is the goal which is the leftmost active (i.e., whose execution has not yet completed) goal which has access to that variable.

Thus, the producer for a real dependent variable changes when its current producer goal completes its execution without rnv-binding the real dependent variable: the next leftmost still-active goal which shares the variable becomes the producer for the variable, and if its execution was suspended on the variable, it can now unsuspend and bind the variable. This second route to unsuspension guarantees that a suspended goal will always be unsuspended *if* goals to its left terminate. That is, if the sequential Prolog program terminates, then unsuspension will always occur.

Note that the two conditions for unsuspending are mutually exclusive, i.e., a task will unsuspend if either condition for unsuspending is fulfilled, but both conditions cannot be fulfilled simultaneously.

So in the example, if `b(X)` was suspended on `X`, then when `a(X)` completes its execution, `b(X)` can unsuspend and rnv-bind `X` (to `1`).

Abstractly, this can be regarded as allowing sequential Prolog execution at all times (the leftmost goal), with a sophisticated form of control that allows and-parallelism such that unifications performed by tasks executing goals to the right do not interfere with the sequential computation.

Note that no synchronization is needed for accessing real nondependent variables. Thus the DAP exploited by DDAS is a natural superset of goal-level IAP: IAP goals are simply those goals which do not contain any real dependent variables. However,

the synchronization *can* be applied to real nondependent variables without affecting the correctness of the execution, although the efficiency can be affected as extra (unneeded) suspensions may occur.

With DDAS, all goals can *potentially* be executed in and-parallel: the necessary condition is that the synchronization scheme is applied to all real dependent variables. There are, however, several complications:

- it may not be the most efficient way of exploiting DAP—e.g., some real dependent variables will introduce suspension without much possibility of parallelism; also, allowing all goals to execute in parallel may lead to much speculative and ultimately useless work being performed.
- whether a particular variable is real dependent or not can change dynamically at run-time. It can be very costly at run-time to determine precisely which variables are real dependent variables.

Annotations are used to tackle both these problems in DDAS: they indicate where parallelism may or may not be exploited, thus allowing and-parallel execution to be avoided where it would be unprofitable; and they are also used to indicate which variables are to be treated as *dependent variables*, i.e., variables for which the producer/consumer synchronization is applied. We use the terms nv-binding, v-bindings, dep-access, non-dep-access applied to dependent variables as the counterparts to rnv-binding, rv-bindings, rdep-access, non-rdep-access applied to real dependent variables, respectively.

For goals executing in parallel to behave correctly, all real dependent variables must be labelled as dependent variables, i.e., the following condition must hold: $\mathcal{R} \subseteq \mathcal{D}$, where $\mathcal{R}$ is the set of all the real dependent variables in the goals executing in parallel, and $\mathcal{D}$ is the set of variables treated as dependent variables. Therefore it is possible to label real nondependent variables as dependent, though this can introduce unnecessary synchronization on these variables, so it is desirable to keep $\mathcal{D}$ as close to $\mathcal{R}$ as possible.

2.2.2. Syntax of Annotation Used For DDAS. Many annotation schemes are possible. The annotation scheme that is developed is an extension of the Conditional Graph Expression (CGE) notation proposed by Hermenegildo (as a refinement to the notation of DeGroot [7]) for IAP [11], and like the CGE notations, they indicate *where* parallelism should be exploited, and not *how* it should be exploited. The annotations can be supplied (transparently to the user) by the system as part of the compilation process, or by the programmer directly.

The extended CGE scheme used is called the *Extended Conditional Graph Expressions* (ECGEs). Like the CGEs, ECGEs are annotations that enclose a set of two or more consecutive body goals, which are to be run in parallel:

```
(<CGE conditions> -> b1(...)  & ... bn(...))
```

The ECGE allows the labelling of dependent variables dynamically. A variable that is labelled as dependent is treated as a dependent variable. This is done by extending the syntax of CGE with an annotation `dep/1` which marks variable(s) (or the unbound variables inside a structure if the annotated variable is bound to a structure) occurring textually inside the CGE as a path to a dependent variable (or variables). These dependent variables obey the producer/consumer rules

already outlined. Variables not annotated by this annotation are assumed to be nondependent.

2.2.3. Tracking Producer Status. In general, an ECGE may contain more than one dependent variable, and tracking the producer status for each individual dependent variable can become quite complex. Thus, in DDAS, dependent status is not tracked for each dependent variable individually; instead, it is tracked at the goal level. In its simplest form (in the next section we shall describe a refinement of this basic scheme), if a particular goal in a producer position for a dependent variable completes its execution without binding the variable, the producer status is passed to the next leftmost goal, which is not necessarily the next leftmost goal which has access to the dependent variable. The advantage of this is that it allows for only one system per ECGE for determining the producer status, which makes keeping track of the producer status simpler; the disadvantage is that the producer status may take longer in some cases to be passed to some dependent variables. However, this basic scheme can be improved, leading to a more precise determination of producer status, by the introduction of the concept of groups. In addition, groups can also improve the backward execution, as shall be discussed in Section 2.4.

2.3. Groups of Goals

One important difference between &-Prolog [12] and DDAS is that the goals being executed in and-parallel are not independent of each other in DDAS. However, it is possible to preserve the concept of independent computation at a higher level of granularity, i.e., "groups" of goals which are executing in parallel and share dependencies can be independent of other "groups" with which they do not share dependencies. As already stated, this concept of "groups" will lead to improved forward and backward execution of DDAS.

If one or more goals (considered together as a group) in an ECGE do not share any dependent variables with some other goals in the ECGE, they are known to be independent of these other goals. More precisely, for an ECGE containing the goals $B_i \cdots B_j$, $j > i$, the goals are formed into one or more groups, with the following conditions defining a group:

- The goals in a group have no variable dependencies with goals in other groups in the same ECGE. That is,

  $$\forall B_x\colon B_x \in G_m, \quad \forall B_y\colon B_y \in G_n, \quad m \neq n, \quad dvars(B_x) \cap dvars(B_y) = \emptyset$$

  where G_n, G_m are groups in the ECGE, B_x and B_y are and-goals, $\emptyset$ is the empty set, and $dvars(B)$ is defined as the set of dependent variables that are passed to the goal B when it is called.

- Two goals which have *direct* or *indirect* variable dependencies with each other must be members of the same group. Direct variable dependency between two goals B_i and B_j is where

  $$dvars(B_i) \cap dvars(B_j) \neq \emptyset$$

  and indirect variable dependency between B_i and B_j is where B_i has direct variable dependency with another goal B_k, $k \neq i$, $k \neq j$, which has a direct or indirect variable dependency with B_j.

```
foo(X,Y,Z) :-
  (dep([X,Y,Z]) -> a(X) & b(Z) & c(X,Y) & d(W) & e(Y) & f(Z) & g(X))
```

FIGURE 1. An example ECGE.

Note that the definition means that it is possible for goals which have no variable dependencies to be in the same group. In the extreme case, all goals are grouped into one group. For most precision, the goals should be divided into as many groups as possible, i.e., where all members of a group have direct or indirect variable dependencies with each other.

Note that goals in a group need not be textually next to each other in the program, and that which group a goal belongs to is determined by how the dependent variables are shared at run-time. An ECGE can organize goals into different grouping configurations based on both conditional tests at run-time on the groundness and independence of variables, and compile-time analysis of the sharing of dependent variables between the goals.

For DDAS, the system must already have information on dependencies of variables in order to extract the parallelism. This information can also be used, without imposing any extra cost, for deciding the grouping of goals. Consider the ECGE example in Figure 1. If the dependent variable `Z` is independent of the other dependent variables,[8] then the and-goals can be divided into three groups: `a(X)`, `c(X,Y)`, `e(Y)`, `g(X)` that shares `X` and `Y`; `b(Z)` and `f(Z)` that shares `Z`; and `d(W)` that does not contain any dependent variable.

With goals classified into different groups, the passing of the dynamic producer status can be made more precise with very little extra cost: the producer for a dependent variable is the leftmost active goal in the *group* the goal belongs to. So for example, for the ECGE of Figure 1, when `a(X)` returns an answer, the producer status is passed to `c(X,Y)` (if `c(X,Y)` is still executing), and not to `b(Z)`, which would be the case in the simple single group per ECGE scheme.

2.4. Backward Execution

In the last section, the synchronization of the parallel execution through dependent variables has been described. However, one important and Powerful aspect of Prolog has not yet been discussed: the backward execution, i.e., the actions performed by the system if a failure occurs, or if an alternative solution is desired. In sequential Prolog, the actions are relatively simple, as there is only one thread of execution: the system restores itself (via backtracking) to a previous state where an alternative exists, and that alternative is tried. However, the actions to be performed in the presence of and-parallelism become much more complex, because the question of "restoring to a previous state" is much more difficult to define, and handling and coordinating the actions of the different and-parallel tasks that are affected can become very complex. These actions constitute the "backward execution," of which sequential backtracking is the simplest case.

With many past proposals of exploiting DAP, the backward execution has either been entirely omitted (as in the committed choice family of languages, e.g.,

[8]This can either be deduced by compile-time analysis, or by a simple run-time test. The test can either be inserted by the programmer or by the compile-time analyzer.

[9, 24, 35]), the and-parallelism restricted (as in [19, 22]), or a very complex (and probably inefficient) scheme is used to handle the backward execution (e.g., [5, 33]). For DDAS, a somewhat different approach is taken.

The necessary condition for a backward execution scheme for DDAS is that correct behavior with respect to Prolog is preserved. That is, the same termination behavior as Prolog is maintained, and the same solutions as Prolog are returned, in the same order. In addition, the effects of any side effects have also to be reproduced correctly. However, beyond the necessary condition, issues such as efficiency and complexity of the scheme should also be considered. One issue is that of speculative and wasted work. In the context of parallel Prologs, speculative work is best defined as work which might not be performed by a sequential Prolog when it is executed, and wasted work as speculative work that is known to be work that a sequential Prolog system would not perform. And-parallelism is inherently speculative: goals are executed in and-parallel ahead of the sequential order, with the expectation that all and-goals that sequentially preceded it would be successful. DDAS's DAP makes the additional speculation that the binding supplied by the producer is correct, i.e., any consumers of the binding proceed on the assumption that the binding generated by the producer is the binding the producer would generate when it succeeds. If, however, after producing the binding, the producer subsequently withdraws the binding on backtracking (without first producing a solution), then the computation performed by the consumer since consuming the binding becomes wasted work. A correct backward execution scheme must discard all wasted work, but in addition, it can also discard some and-parallel work that is not wasted work without affecting correctness, *as long as* such work is eventually (re)performed. The reason why it may be desirable to discard nonwasted and-parallel work is that it is very difficult to keep precise track of what is wasted work. In general, this would imply keeping a close track of the forward execution, which would impose very high run-time overheads. In addition, the backward execution scheme is likely to be very complex, which would impose more overhead, and be more difficult to ensure correctness. It is far cheaper and simpler to have a scheme that is less precise in keeping track of wasted work, and which may then discard some nonwasted work.

The scheme currently used for DDAS is based on that proposed for &-Prolog [11], with suitable extensions to deal with DAP and the concept of groups. In particular, the concept of groups allows a similar form of intelligent backtracking as &-Prolog at the level of groups of goals. As in &-Prolog, there are two basic situations that can occur during backward execution:

Outside backtracking: This occurs after all the and-goals in the ECGE have succeeded, and execution has continued to goals following the ECGE. Outside backtracking is the situation when the system then backtracks into the ECGE from the outside. In such a case, the backward execution behaves much as in sequential Prolog, with the alternative in the rightmost and-goal in the ECGE with an alternative being selected, and forward execution started again.

Inside backtracking: This occurs with a failure in one of the and-goals, before all the and-goals in the ECGE have succeeded. Here, the concept of groups comes into play. There are two situations:

- During backtracking, a binding to a dependent variable is undone. If this binding has been consumed by other and-goals, the work performed on

those and-goals becomes wasted work. For simplicity, if the binding has been consumed (this can be recorded at a very low cost using a bit flag), then all the work done on the and-goals *in the same group* to the right, and which consumed *any* dependent variables' binding, is discarded. As all the work done by these and-goals is discarded, their work has to be redone; and this can be done at any time after the work has been discarded. In the current scheme, the and-goals are allowed to restart immediately after the wasted work is discarded: this maximizes parallelism but is also the most speculative approach, as the restarted goals can be discarded again. It is expected that and-parallel executions which would lead to repeated discarding of the consumer's work would be prevented from executing in and-parallel by the ECGE annotations.[9] For the example ECGE of Figure 1, if inside backtracking in `b(Z)` undoes a binding to `Z`, then work done on `f(Z)` will be discarded if it has consumed any dependent variable bindings (not necessarily that of `Y`).[10] The execution of `f(Z)` would then be allowed to restart immediately—if a worker is free, then the goal would be restarted.

- A failure causes backtracking all the way to the beginning of an and-goal. In sequential Prolog, backtracking would continue into the goal immediately to its left. However, for DDAS, the concept of groups allows a simple form of intelligent backtracking: backtracking can continue in the closest goal in the same group that is to the left, thus skipping any goals that are in other groups (e.g., in the example ECGE, `a(X)` is the goal to backtrack into if the system inside-backtracks to the top of `c(X,Y)`), as long as the goals being skipped contain no side effects. If there are side effects, then the backtracking would have to be to the goal to the left.

  Another problem that does not occur in sequential Prolog is that the goal that the system wants to backtrack into may not yet have completed its execution, as it is executing in parallel. In this case, that goal will fail immediately when it does complete. This can be implemented by setting a "to_fail" bit flag that is associated with the and-goal. Note that if the unbinding of a dependent variable causes the original backtracking and-goal to be discarded, then the "to_fail" flag is also reset. The reason why the and-goal (`a(X)` in the example) cannot be told to fail immediately if it has not yet completed is because computation to the left may change the bindings of dependent variable(s), which would then invalidate the computation of the failed and-goal (`c(X,Y)`).

- A special case of the previous case is when the system inside-backtracks to the top of an and-goal that is also the leftmost and-goal of a group (e.g., `a(X)`, `b(Z)`, and `d(W)` in the example). In this case, another opportunity for a simple intelligent backtracking arises: the whole ECGE can fail immediately, because none of the computation in the ECGE can "cure" the failure of such a goal. Of course, none of the goals in the ECGE

[9] A dependent variable may be dependent in more than one level of ECGE. In this case, the discarding and redoing of work when a binding is undone occurs in the ECGE in which the dependent variable first became dependent. This automatically ensures that the work done in the later ECGEs is undone as well.

[10] If an and-goal contains its own ECGE, then it would be marked as having consumed dependent variables. This is done to avoid the need of checking and setting more than one flag.

should contain any side effect. If there is side effect, then the system must backtrack into the goal to the left.

This backward execution scheme has the advantage of being quite simple: not much overhead is imposed on the forward execution in order to make the backward execution more precise. The backtracking can discard quite a lot of computation that is not necessarily wasted work, but experimental evidence suggests that this can be minimized or even eliminated entirely if the dependent variables are not bound "shallowly," i.e., they are bound after the goals which act as "guard" goals. As this is generally good programming practice in any case (for example, see [20]), the imprecise tracking of wasted work is not a problem.

It is not entirely clear how useful the concept of groups is for real-life Prolog applications; the scheme has so far only been tested with simple artificial examples, for which it does show a significant improvement [29]. The current compiler for DASWAM generates codes that assume only one group per ECGE, although the support for groups has been implemented fully in DASWAM. It is expected that the scheme will become more useful when compile-time tools for automatic annotation of ECGE are developed, as these tools may produce more ECGEs than a human would. This may lead not only to unexpected parallelism, but also more problematic executions than programs annotated by hand. Here, the concept of groups may be very useful. In any case, there is little harm in providing the concept of groups, as the overhead for doing so implementation-wise is extremely low.

2.5. An Example

It is beyond the scope of this paper to fully demonstrate the utility of DDAS. Instead, a simple example will be given here to illustrate the exploitation of nondeterminate DAP, a form of parallelism that has not yet been effectively exploited in other parallel Prolog systems. Consider the program in Figure 2, where, given a list of persons, `People`, and a person, `Person`, then `languages_of_sgs/4` finds from `People` the people who are "cousins of the same generation"[11] of `Person`, and from these cousins, produces the list `Speakers` of people who speak language `L`. The predicate has as many solutions as there are languages in the database, and as it is possible for a person to be able to speak more than one language, so the program has nondeterministic and-parallelism.

DDAS, with the ECGE annotations shown, can execute this program in and-parallel: `find_sgs/3` generates the list of cousins, `Sgs`, which is incrementally consumed in parallel by `all_speak/3`, generating the list of speakers for the first language, and subsequently the lists for other languages can be generated through backtracking.

This program will also benefit from or-parallelism: the lists of speakers for the different languages could be generated in or-parallel. DDAS can be readily extended to exploit or-parallelism: a proposal to do so, based on the "or-under-and" method of combining IAP and or-parallelism [30], is called Prometheus [27], and is able to exploit and-(IAP and DAP) and or-parallelism. As each branch of the search-tree is computed independently in the "or-under-and" method, the addition of dependent and-goals adds no extra complications to the conceptual model.

[11] The `sg/2` predicate, which defines "cousins of the same generation", is taken from Ullman [36, p. 797].

```
languages_of_sgs(Person, People, L, Speakers) :-
   (dep([Sgs,L]) ->
       find_sgs(People, Person, Sgs) &
       language(L) &
       all_speak(Sgs, L, Speakers)
   ).

find_sgs([X|People], Person, Sgs) :-
    sg(Person, X), !, Sgs = [X|Sgs1],
    find_sgs(People, Person, Sgs1).
find_sgs([_|People], Person, Sgs) :-
    find_sgs(People, Person, Sgs).
find_sgs([],_,[]).

sg(X, X) :- person(X).
sg(X, Y) :-
    parent(X,Xp), sg(Xp, Yp), parent(Y, Yp).

all_speak([X|Sgs], L, [X|Speakers]) :-
    speak(X, L), !,
    all_speak(Sgs, L, Speakers).
all_speak([_|Sgs], L, Speakers) :-
    all_speak(Sgs, L, Speakers).
all_speak([],_, []).

/* followed by database of people, parent, language, speak */
```

FIGURE 2. Example program with nondeterministic DAP.

3. OVERVIEW OF IMPLEMENTATION OF DASWAM

DASWAM is an implementation scheme for DDAS. In this scheme, workers correspond to Prolog engines, each of which is a WAM modified so that it can cooperate with other workers to execute a program in dependent and-parallel. In the following sections, various aspects of DASWAM are presented.

In the next section, the memory organization of a DASWAM worker is overviewed: some understanding of this aspect of DASWAM is needed to understand its implementation, but as much of this is not unique to DASWAM, the treatment is relatively brief. In Section 4, many of the most novel aspects of DASWAM that are needed to deal with DDAS's DAP are described. In Section 5, the DASWAM instruction set, which can be considered as the "glue" that coordinates the various parts and features of DASWAM into executing a Prolog program in parallel will be discussed; the instruction set is an extension of the standard WAM instruction set, so due to the limitation of space, the discussion will concentrate on the extensions. Finally, the implications of handling the impure features of Prolog, to both DDAS and DASWAM, will be discussed.

3.1. Memory Organization

Like many implementations of parallel Prolog (e.g., [2, 10, 11, 15]), DASWAM adopts a "distributed stack scheme" [11], where each worker maintains a **stack set**, consisting of the normal Prolog stacks (plus some extra data areas to handle

the special requirements of DDAS). The stacks are used in much the same way as normal stacks during the execution of a task, but when a task finishes or suspends, new tasks can be started for the worker, using the same stacks by using the space on top of the space used by the older tasks. Each area in the stack used by a particular task is referred to as a **section**,[12] and sections are separated from each other by **markers**. This is an adaptation of the marker scheme of Hermenegildo [11].

Like &-Prolog, DASWAM retains the environment stacking model of the WAM, and is thus somewhat unlike the goal stacking models adapted by some schemes that exploit and-parallelism, such as JAM Parlog [6] and Andorra-I [22]. Like &-Prolog, there is a "goal stack" for each worker, used to store the pointers to the available and-parallel goals generated by the worker, and which can be "stolen" by other workers to start new tasks. The memory organization of a worker is shown in Figure 3. A complete description of a DASWAM worker is outside the scope of this paper, but some relevant features are: the local stack is split into the

[12]This was referred to as a stack segment in [27, 28], but stack section is used here to be consistent with terminology used for &-Prolog.

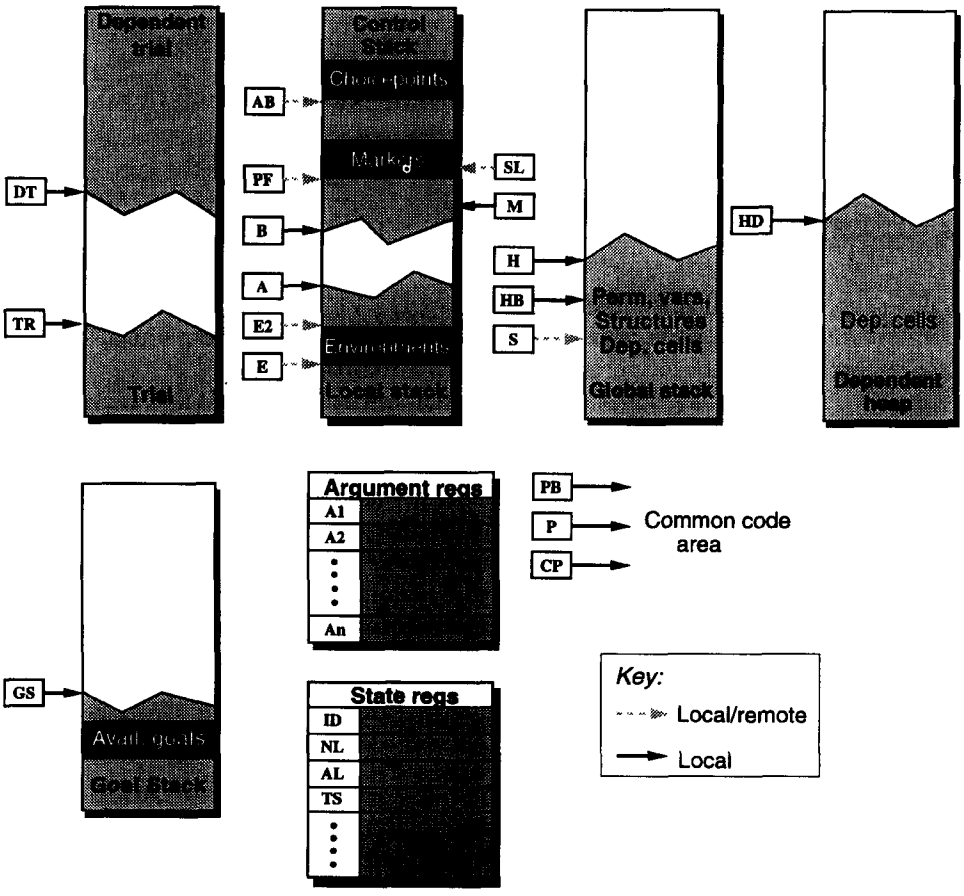

FIGURE 3. Storage model for a DASWAM worker.

environment and the control stack, as in SICStus WAM [3]. The control stack is used to store the choice-points and markers. Markers are used to separate stack sections, but they occur in the control stack only, with the separation of sections in the other stacks indirectly indicated by pointers in the corresponding markers. Some of the extra data areas not found in the WAM include the goal stack, the dependent trail, for trailing the bindings to dependent variables (all bindings to dependent variables have to be trailed, as special actions might have to be taken when such variables are detrailed on backtracking, see [26, 27]), and the dependent heap, used for storing some of the dependent variables. The dependent heap is managed as a true heap (i.e., space is not recovered directly by backtracking). However, unlike the implementations of committed choice languages, not all variables, or even all dependent variables, are placed on the dependent heap. Thus, the problem of garbage collection, while still very important in DASWAM, is not as critical as in the committed choice languages, and the prototype does not have a garbage collector. In addition to the stack set, a DASWAM worker also consists of a set of state registers, an extension of the set used by sequential WAM, and a set of argument/temporary registers. Figure 3 shows the general location that some of the state registers can point at. Darker arrows mean those registers can only point into the worker's own stack, and lighter dashed arrows mean those pointers which can point into some other worker's stack set.

3.2. Organization of Markers

Markers separate stack sections, and guide the management of the stack sections. In addition, depending on the nature of the sections above or below them, some markers may serve some additional special functions. There are five basic types of markers:

Parcall marker: This marks the start of an ECGE, with the following stack section representing the leftmost task. The marker is used to coordinate the activities of the ECGE, and is very similar to the P_Call Frame of &-Prolog, except that it is allocated on the control stack and not on the environment stack as in &-Prolog, and is just treated as one type of marker. Each and-goal in the ECGE is represented as a slot in the parcall marker. The slot contains such information as the status of any task executing that and-goal, and pointers to the marker marking the start and end (when completed) of the task associated with the and-goal.

Join marker: This marks the end of an ECGE, and is allocated by the worker completing the last task of the ECGE, following its completion. This is similar to the wait marker of RAP-WAM [11], but is more flexible in that it can be allocated by a different worker from the one that allocated the parcall marker for the ECGE, as that worker does not have to wait for the completion of the ECGE as in RAP-WAM. The section following this marker represents the work immediately following an ECGE. It also contains pointers to the rightmost task of the ECGE, to allow backtracking into an ECGE.

Suspend marker: This marks the suspension of a task on the previous stack section. When woken, the task can then be continued in another location in the distributed stack. This marker is allocated when the worker that was

working on the suspended task picks up a new piece of work. The suspend marker contains the state of the worker when it was suspended: this is used to set the state of the worker which resumes the execution of the task when it becomes unsuspended.

Continuation marker: This marks the continuation of a task following the marker which started in a different location in the distributed stack. This marker contains pointers to the previous stack section of the task before it was suspended, to allow backtracking across the stack sections.

Marker: This is the basic marker, marking the start of a section that is not of the above types, e.g., when a new task is started. This corresponds to the input goal marker and the local goal marker of RAP-WAM.

3.2.1. Structure of Markers. The general structure of a marker is shown in Figure 4. A marker is divided into three general parts: (i) a *management* part for managing the marker and sections; (ii) a *state* part, which stores the pointers to the various stacks, and the values of some other state registers when the marker was allocated, thus serving to separate the stack sections[13]; and (iii) a *special* part, which serves the special requirements of the various types of markers (and is empty for the basic markers).

[13]The amount of information that has to be stored in the state part is purely for separating the stack sections, and is insufficient to allow a suspended task to resume execution; extra information is needed for this, and is stored in the special part of a suspend marker.

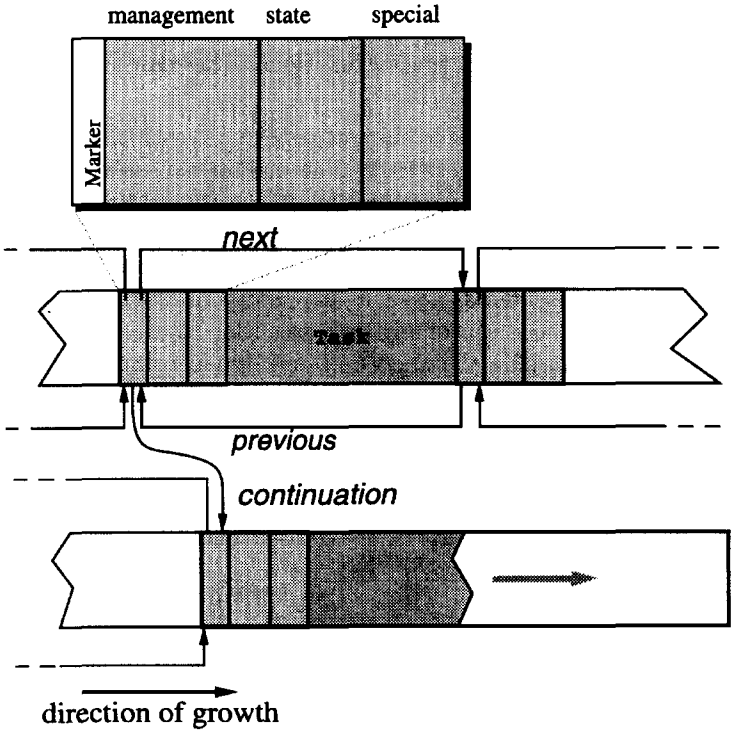

FIGURE 4. Structure of a marker.

Markers are linked to the next and to the previous markers in the same stack set by the next and previous marker pointers. If the marker is the topmost (most recent) marker, the next marker pointer is not set. A third pointer, the continuation pointer, is set to point at the continuation marker that continues the task following the marker on another stack section, and is not set if the section has no continuation. The continuation worker field is the ID of the worker that continued the task. This arrangement allows a task to be followed in both directions through the distributed stack. Figure 4 shows a task (shaded gray) started in the top stack section, and continuing in the bottom stack section.

4. DEALING WITH DDAS'S DEPENDENT AND-PARALLELISM

To implement DDAS's DAP, two main features of DDAS have to be dealt with: that of the dependent variable, and the suspension/unsuspension mechanism. These aspects are discussed in this section, and in addition, the mechanism needed to implement the groups concept is also presented.

4.1. Implementing the Dependent Status

The dependent status is implemented using dependent cells. A dependent cell can be thought of as a special type of value cell. It indicates that the term represented by the value cell should be treated as dependent. Dependent variables are the means by which DDAS synchronizes the parallel execution of tasks with dependencies, and on an implementation level, they can be regarded as variables for which special actions may have to be performed when a binding is attempted, or when a binding is undone during backtracking. Thus, the dependent cell allows DASWAM to recognize that such actions have to take place. In addition, the dependent cell carries the information DASWAM needs for handling suspensions/unsuspensions.

In an ECGE, the `dep/1` annotation is used to annotate a particular textual variable as dependent. In implementation terms, this means that any accesses through the annotated variable would be treated as dependent in the ECGE.[14] If an annotated variable dereferences to a term containing any unbound variables, those variables will obey the producer/consumer relationship within the ECGE.[15] A term can contain unbound variables either because it is an unbound variable, or because it is a compound term which contains an unbound variable. A term that is an unbound variable is converted to a dependent variable by binding it to a newly created dependent cell. One way to treat a compound term is to traverse the whole term exhaustively, converting all unbound variables into dependent variables. This can be a very expensive operation, especially if the term is large and, unless groundness information is somehow available, a compound term with no unbound variable, which may be quite common, still has to be traversed. An alternative way is to convert compound terms lazily, i.e., unbound variables in such terms

[14]Note that if the dependent variable is aliased to another variable before entering the ECGE, and both variables occur textually inside the ECGE, then both should be annotated with `dep/1`, as they represent different paths of access to the term represented by the dependent variable. If a variable is aliased to a dependent variable inside the ECGE, then this represents the same path of access, and only the dependent variable has to be marked as dependent.

[15]Suspension will occur if the term already contains any dependent variables in a consumer position.

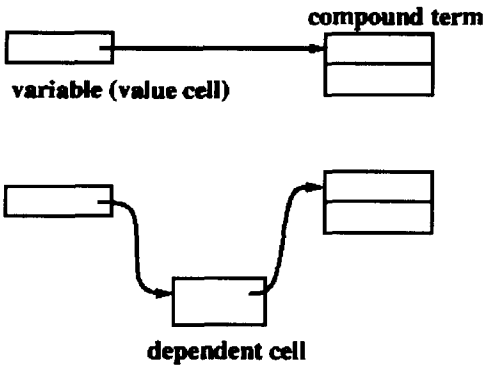

FIGURE 5. Making a compound term dependent.

are turned into dependent variables on demand, when they are actually accessed. The compound term must be recognized as dependent in order to allow this lazy conversion of dependent variables. This is done by inserting a dependent cell into the reference path to the compound term, as shown in Figure 5.

Such conversions must be undone when the system backtracks out of the ECGE. This is done by "trailing" the original value, so that the original unconverted value can be restored during backtracking.

There may be more than one reference path to a dependent compound term through aliasing. Such aliasing can be classified into four types:

1. The aliasing was done before the ECGE was entered.
2. The aliasing was done inside the ECGE, but before the term became dependent.
3. The aliasing was done inside the ECGE, after the term became dependent.
4. The aliasing was done after exiting the ECGE.

In case 4, the compound term is no longer dependent, so aliasing is not an issue. In case 3, the newly aliased variable can be made to point to the dependent cell. The situations for cases 1 and 2 are more complex, as the compound term may be accessed through alternative reference paths, without going through the dependent cell.

For case 2, if the newly aliased variable is nondependent itself, then it can only be accessed by the local and-goal directly. If the dependent term is in a producer position, it is safe for the and-goal to bind the term's unbound variable. As shall be discussed in the following sections, the unification mechanisms of DASWAM ensure that any reference to such dependent variables in a consumer position is properly detected.

Case 1 is dealt with by requiring that all paths to a dependent variable at the start of an ECGE be marked separately, that is, they should all be marked by the `dep/1` annotation.[16]

4.1.1. Unification Mechanisms. In the WAM and DASWAM, a unification can be compiled into one or more abstract machine instructions. Such instructions in

[16]Note that this does not alter the basic requirement, i.e., all variables that cannot be shown to be independent must be labelled as dependent. It also does not require the system to recognize that particular dependent labellings are actually to aliased variables. What the requirement does imply is that just because two variables are aliased, one cannot assume that marking one as dependent will automatically mean the aliased variable would be recognized as dependent.

the DASWAM are designed to insert the dependent cell where appropriate. Some extra mechanism (with respect to the WAM) is needed to deal with the case of unification of a compound term that appears textually. For example, consider the goal `L = f(W)`. This will be compiled into something like:

```
get_structure f/1, A0
unify_value A1
```

In this case, if `L` is dependent, unbound initially, and in a producer position, `W` must also become dependent when it is referenced by `unify_value`. Thus, the fact that `L` is dependent must be retained when `unify_value` is executed. This is done by storing the pointer to `L`'s dependent cell in the DASWAM register D1. A `unify` instruction will first check if `D1` is null. If it is, the normal actions of the instruction take place. If it is not null, then the actions dealing with dependency have to be performed. The management (i.e., nonvalue cell) part of a dependent cell for `W` will be much the same as that for `L`, as they have the same producer/consumer status; thus `L`'s dependent cell is used as a template to construct that of `W`'s.

In the (DAS)WAM, many unifications can be broken down and specialized to specific abstract machine instructions; but the unification routine may need to be invoked in some cases. In such cases, the two terms being unified may contain dependent terms, where any variables appearing as subterms of the dependent terms must be marked as dependent. In the DASWAM, two registers are provided for these purposes, one for each term being unified. These are the `D1` and `D2` registers. They store any dependent cell encountered to act as a template for constructing new dependent cells.

4.1.2. Lazy Conversion. A potential performance problem exists when an initially nondependent compound term becomes dependent, i.e., when a dependent variable is bound to it. The compound term may contain unbound variables, which now become dependent. Again, these variables are converted lazily.

Consider the clause shown in Figure 6, where the clause is called with `D` being a dependent variable in a producer position. If `D` is initially uninstantiated, as shown in Figure 6(a), with the uninstantiated dependent cell represented in gray, then when `D` is unified with `L`, `L` is changed to point at `D`'s dependent cell, and the dependent cell set to point at `f(W)`, as shown in Figure 6(b). `W` is left unconverted because variables are only converted lazily.

If the above clause is called with `D` in a consumer position, then when `D = L` is executed, if `D` is unbound, the need for suspension will be detected, as `D` is known to be dependent via its dependent cell. If `D` is bound, say to `f(X)`, then the unification routine would be invoked, with `D` known to be dependent. So when the unification routine tries to unify the two subterms, `X` and `W`, then if `W` is not a dependent variable, the unification routine will mark it as dependent. Thus, when the consumer tries to bind `W` to `g(Z)` in the goal `W = g(Z)`, suspension occurs. This situation is shown in Figure 6(c), after the suspension, with `W = g(Z)` succeeding.

In these cases, a new dependent cell can be created by the consumer to allow for suspension. The dependent status of the variable (`g(Z)` in the example) is not invalidated by backtracking of the consumer, as other goals may have access to the variable. Thus, the dependent cell is placed on the dependent heap and not the global stack, so that it would not be destroyed by backtracking. Space can only be recovered on the dependent heap by some form of garbage collection.

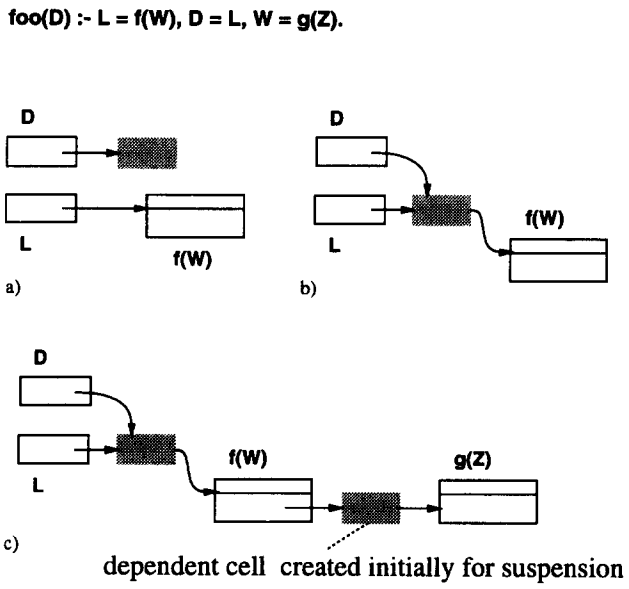

FIGURE 6. Example of lazy conversion.

4.1.3. Binding a Dependent Variable to a Dependent Variable. The definition of nv-binding means that a dependent variable can only bind to another dependent variable if they are both in a producer position. The reason for this is that, in general, a dependent variable in a consumer position should not be bound to another term. Although in theory it is possible to allow a dependent variable in a producer position to bind to a dependent variable in a consumer position under certain circumstances, it would be very difficult and complex to determine the producer/consumer status of the dependent variables.

With the restrictions, suspension will occur if either dependent variable is not in a producer position. However, although the variables are bound in a producer position, they can be accessed from a consumer position for either variable, and a further complication is that the dependent variables may be created in different ECGEs. For correct behavior, the correct dependent cell (and hence the correct ECGE) must be accessed to determine the producer/consumer status. This is illustrated by the example program fragment of Figure 7: the two dependent variables (`X` and `Y`), created in different ECGEs are bound together inside `c/2` by `X = Y`. As both dependent variables are in producer positions, the binding is allowed. Now, consider

```
foo :- (dep(X) -> a(X) & b(X)).

a(X) :- (dep(Y) -> c(X,Y) & d(Y)).

c(X,Y) :- X = Y, X = 1.

b(X) :- X = 2.

d(Y) :- Y = 3.
```

FIGURE 7. Example binding of dependent variables.

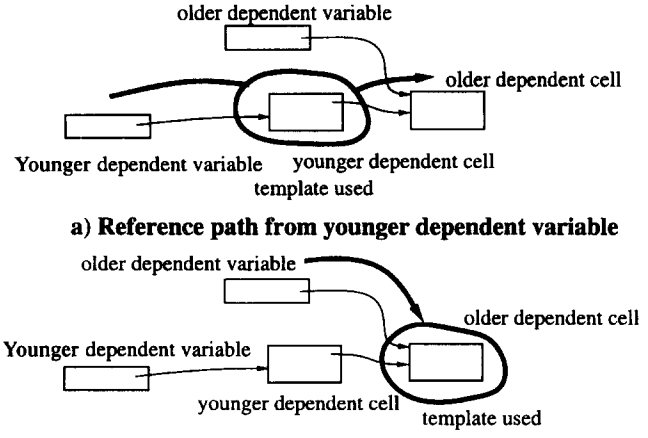

FIGURE 8. Binding two dependent variables.

the three accesses to the dependent variables in `b/1` (`X = 2`), in `c/2` (`X = 1`), and in `d/1` (`Y = 3`), assuming that the access occurs *after* the two dependent variables have been bound together. For `b/1` access, the correct dependent cell to check the dependent status is `X`; and for both the `c/2` and `d/1` accesses, it is `Y`.

Because of the restrictions on binding of dependent variables, it is possible to ensure the correct behavior by following a simple convention for the binding of two dependent variables: when a dependent variable is bound to another, the value cell of the younger dependent variable (the one which is created in a younger ECGE—which must, by definition, be enclosed by the older variable's ECGE for both dependent statuses to be valid), is set to point at the older dependent variable (thus, in the example, `Y` would be bound to `X`). When dereferencing a variable, if more than one dependent cell is encountered, the one that is youngest is used as the template and also for determining the producer/consumer status, as shown in Figure 8. Thus, if accessed from the older dependent variable, its dependent cell is the only one encountered, and it is the one used; but when accessed from the younger dependent variable, both dependent cells are in the reference path and only the younger dependent cell is used.

Note that this convention ensures that when a dependent variable is bound to a nonvariable, the dependent cell that is used for storing the binding is the oldest dependent cell. Thus, if the dependent binding is later undone during backtracking, any associated effect (e.g., redoing of sibling and-goals to the right, as discussed in Section 2.4) is performed on the oldest ECGE in which the variable became dependent, as is required.

4.2. Dealing with Suspensions

4.2.1. Suspending a Task. In DDAS, a task can suspend in the middle of a unification. As unifications can be represented by several DASWAM instructions, this means that suspensions can occur in many DASWAM instructions, and in various places during the execution. In order to better utilize resources, a worker that was working on a suspended task can perform other work if such work is

available. In such cases, a suspend marker is allocated on the stack, containing the state of the worker so that the state can be restored when the task is resumed. As the suspension can occur almost anywhere, all the registers, including those temporary registers that are currently in use, have to be saved. A new state register, the "number of temporaries" register (TS), is used to record the maximum number of temporaries and argument registers that are used inside a particular clause body. TS is set when a clause is entered (the compiled code is modified to provide this information about the maximum number of temporaries), and is used to determine the number of argument/temporary registers that will be saved if suspension occurs. If less than the maximum number of registers are actually used at the point of suspension, then some space is wasted. There is therefore some scope for improving on this basic scheme, such as updating TS after returning from a procedure call.

4.2.2. Dealing with Unsuspensions. The implementation of unsuspension of tasks consists of the following steps:

- detecting that a condition for unsuspension has been fulfilled;
- locating the suspended tasks that are unsuspended;
- resuming the execution of the suspended tasks.

DETECTING FULFILMENT OF UNSUSPENSION CONDITIONS. As discussed in Section 2.2.1, there are two conditions for unsuspending a task:

- when the dependent variable becomes bound to a nonvariable by the producer of the variable,
- when the suspended task becomes the leftmost uncompleted task in its group.

The first condition for unsuspending a task is very similar to that found in many other suspension mechanisms, such as delay primitives in Prolog systems like MU-Prolog [18] and the committed choice languages. The condition is fulfilled whenever a dependent variable is bound, and so is easily detected.

The second condition is probably unique to DDAS, and requires the detection of when tasks become leftmost. The leftmost status of a task, changes *only* when the task executing the current active leftmost goal in a group finishes: the tasks executing the next leftmost goal in the group (there may be more than one task, as the goal may have forked into several tasks in nested ECGEs) become in the producer position for dependent variables in the group. Thus, when a task completes the execution of an and-goal, it checks to see if it was the leftmost active goal for the group: this can be easily done by examining the parcall marker associated with the ECGE where the current statuses of and-goals in the ECGE are recorded. If it was the leftmost active goal in the group, then the second condition for unsuspension is fulfilled.

LOCATING SUSPENDED TASKS TO UNSUSPEND. Once an unsuspension condition has been detected, suspended tasks for which the condition was fulfilled need to be located so that they can unsuspend. This is done by maintaining a list of suspended tasks for each suspension condition such that when the condition is fulfilled, the tasks in the list can be unsuspended. As each task can unsuspend on two conditions, each task occurs in two lists: one to allow the task to unsuspend when the dependent variable it was suspended on becomes bound, and another list to allow the task to unsuspend when the task becomes leftmost.

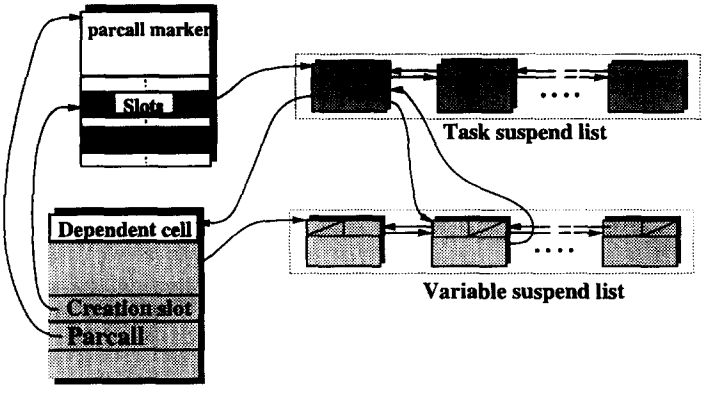

FIGURE 9. Mechanism for suspension.

Thus, each dependent cell, before it becomes bound, maintains a *variable suspend list*: each time a task suspends on trying to nv-bind the variable, the task is suspended, and a suspend cell, which contains information on the task—the ID of the worker working on the task before it suspended, and a pointer to the relevant suspend marker (if any)—is added to the variable suspend list. When the dependent variable is bound, the variable suspend list contain the tasks that needed to be unsuspended. This mechanism for dealing with the unsuspension is similar to many previous approaches, such as that taken by JAM Parlog [6].

The second unsuspend condition is unique to DDAS, and is handled by maintaining a task suspend list for each and-goal of the ECGE in which the dependent variable was created. The task suspend list consists of a list of task suspend cells. Each task suspend cell represents a task that has suspended on a dependent variable of the ECGE while executing that particular and-goal (more than one task can be working on an and-goal, as an and-task can be split into subtasks in some nested ECGEs).

The structure used for suspension is shown in Figure 9. A suspended task is represented by a suspend cell and a task suspend cell. These two cells contains pointers to each other, so that when the task unsuspends, both cells can be removed. For example, if the task is unsuspended because it became the leftmost task, then all the task suspend cells in the task suspend list are woken up, and the suspend cell associated with the task has to be removed from the suspend list. Both lists are doubly linked to facilitate the removal of individual elements. The dependent cell contains a pointer to the suspend list, and pointers to the parcall marker representing the ECGE where the variable became dependent, and the first occurrence of the variable (supplied by the compiler) in that ECGE.

RESUMING UNSUSPENDED TASKS. When a task is unsuspended, it can be in one of two states:

- The original worker may not yet have picked up new work after the suspension and thus have not allocated a suspend marker. In this case, the original worker can simply resume execution of the task.
- The original worker may have picked up new work since the suspension and have thus allocated a suspend marker. In this case, the task is added to a pool of unsuspended tasks. A worker that is looking for work can then

pick up the task and resume its execution. The worker would read in the suspended state from the suspend marker, allocate a continuation marker on its own stack, and resume the task's execution.

4.2.3. Finding the Correct And-goal in an Ancestral ECGE. In order to properly suspend, the proper task suspend list to add the task suspend cell has to be found: this is the task suspend list of the goal in the ECGE in which the dependent variable causing the suspension became dependent. There is a complication, as the ECGE the task is currently working on can be a descendent of the ECGE in which the dependent variable is created, and although the correct ECGE can be located via the dependent cell, the actual goal the task is working on would not be immediately known if the task is working on a descendent ECGE, i.e., the goal has split into subtasks. To find the position in the ECGE in which the dependent variable was created, each parcall marker contains a pointer to the parcall marker's parent parcall marker (PP), and a pointer to the slot representing the parent and-goal in which the current ECGE was encountered (PS), as shown in Figure 10. The correct and-goal to place the task suspend cell could be found by following the PP pointer from the current parcall marker until PP points to the parcall marker pointed to by the dependent cell. The PS marker would then be pointing at the correct and-goal.

The cost of location of the correct and-goal is potentially unbounded, but will only be a problem if a dependent variable is passed down many ECGEs (without

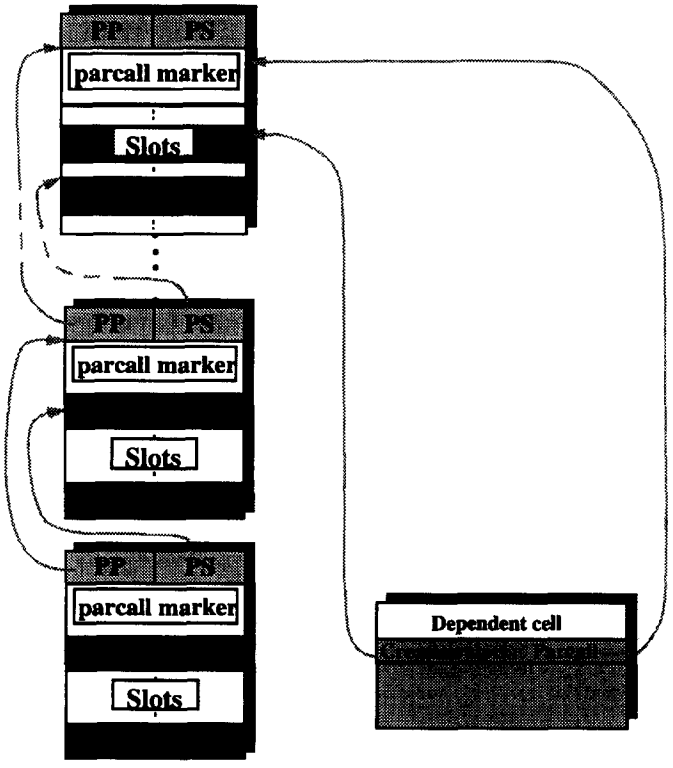

FIGURE 10. Finding the correct and-goal.

being made dependent locally), and bound many nesting levels away from the ECGE that created it. It is expected that, in general, the dependent variable will be bound (or made a dependent variable locally) not too far below the ECGE in which the variable was created.

4.2.4. Determination of Producer/Consumer Status. A reasonably efficient mechanism must be devised to determine the producer/consumer status of a task for a particular dependent variable. As already discussed, the producer for a particular dependent variable at any time is the leftmost still-active goal in the group that is not to the left of the leftmost occurrence of the variable. As in the case for finding the correct ECGE for suspension, the dependent variable may be created in an earlier (ancestral) ECGE. The first problem is thus to know the position the task is in with respect to the ECGE in which the dependent variable was created. This is simple if the ECGE is the current ECGE, but if it is some ancestral ECGE, then the same mechanism as that shown in Figure 10 is used to find the task's position in the ECGE. Once the ECGE is found, then the current producer position for the variable has to be found; this can be done by scanning the statuses of the and-goals in the ECGE. The cost of doing this is quite small, as the number of goals in an ECGE is limited. The scheme that is used in DASWAM is an optimization of a scheme that straightforwardly scans every goal in the group, and is described in more detail in [27, 28].

4.3. Implementation of Groups

Groups are supported in DASWAM by providing pointers in the slots representing goals in the parcall marker: two pointers per slot are needed, to link to the left and right. The pointers are then set up by DASWAM abstract machine instructions when the parcall marker for the ECGE is allocated.

The example ECGE of Figure 1 would be represented as shown in Figure 11 (assuming Z is independent of X and Y). The three groups in the ECGE are labelled as α, β, and γ in the figure—this information is provided to help the reader understand the figure, and is not stored in the slot. The first column (#) of numbers in the figure is the slot number, the second column is the left link (l) and the third column is the right link (r). If there are no and-goals to the left and right respectively, the link is set to the null value, represented as $\oslash$. Thus, groups are formed by appropriately setting the left and right links in all the slots. This is done by several parallel control abstract machine instructions.

#	l	r	group	foo :- dep([X,Y,Z]) ->
0	$\oslash$	2	α	a(X) &
1	$\oslash$	5	β	b(Z) &
2	0	4	α	c(X,Y) &
3	$\oslash$	$\oslash$	γ	d(W) &
4	2	5	α	e(Y) &
5	1	$\oslash$	β	f(Z) &
6	4	$\oslash$	α	g(X).

FIGURE 11. Groups in par-call marker slots.

5. THE DASWAM INSTRUCTION SET

The DASWAM instruction set is largely the instruction set of SICStus' WAM [3], and incorporates many of the ideas from RAP-WAM [11], although the parallel control instructions are different in some aspects. To deal with DDAS, some of the instructions have to be modified, and some new instructions introduced. This section will concentrate on describing those instructions that control the and-parallel execution and construct the representation of groups: the part that shows the greatest difference from the sequential WAM.

In addition to implementing the concept of groups, the instruction set outlined here allows for more flexible compilation of ECGE/CGEs compared to schemes such as RAP-WAM [11]. The instruction set is in fact somewhat smaller than that of RAP-WAM, and the only additional instruction needed for DAP is the instruction to label a variable as dependent.

5.1. Parallel Control Instructions

allocate_pacall(N, r, M)

This instruction represents the start of execution of an ECGE. It allocates a parcall marker at the top of the worker's control stack. The parcall marker contains N slots, where N is the number of and-goals in the ECGE.

The second argument of the instruction, r, represents the position of the and-goal in the same group which is to the right of the leftmost and-goal in the ECGE. This leftmost and-goal is always executed locally by DASWAM.

M represents the number of active permanent variables in the current environment when the parcall marker is allocated. This is needed for correctly calculating the new top of local stack.

Some initializations are performed by this instruction. In addition, the leftmost slot of the ECGE is also initialized. The other slots will be initialized by the instructions that follow the allocate_pcall instruction.

push_goal(C, s, l, r)

This instruction is used to push a goal element, representing work for the s-th and-goal in the current ECGE, onto the goal stack. It also initializes the slot representing the and-goal. This instruction follows allocate_pcall, with each slot, except the leftmost, represented by one push_goal instruction.

C is the address for the code for the and-goal: this is the point to jump to start execution of the and-goal. l and r are the left and right group neighbors for the slot, respectively.

A goal element, consisting of pointers to the current parcall marker and current slot (i.e., slot s), is pushed onto the goal stack. The slot itself is initialized to reflect that this new and-goal is available for pick up.[17]

Both allocate_parcall and push_goal are very similar to their counterparts in RAP-WAM. The next instruction, however, is the main difference between how RAP-WAM and DASWAM handle the management of parallel goals.

[17]This idea of pushing pointers to a goal instead of the goal itself (as is done in RAP-WAM [11]) was suggested by Hermenegildo, and implemented in PWAM (personal communications with M. V. Hermenegildo, 1990), the successor to RAP-WAM.

par_proceed(E, s)

The code for initializing the call to an and-goal is represented by a series of put instructions, followed by a call instruction to call a body goal. This is exactly like a normal call in sequential WAM. The only difference from sequential WAM is that the call to an and-goal is followed by a par_proceed instruction. As in a sequential WAM, when the machine returns from executing the and-goal, the next instruction, which in this case would be the par_proceed instruction, would be executed. This instruction can be considered as the instruction that interfaces the worker to the parallel scheduler: it performs the management function associated with the return of an and-goal; for example, checking if it has been told to fail, checking if it is the last goal in the ECGE to complete (if it is, the worker will execute the code following the ECGE; if it is not, the worker is then free to select other available work to work on).

Using the normal call instruction to implement a call to an and-goal, instead of executing all and-goals via a special instruction as in RAP-WAM [11], has two important advantages:

- The compilation of DDAS will be very similar to that of sequential WAM. This makes the writing of a compiler much simpler. In fact, the DASWAM compiler was modified from SICStus' compiler without too much effort.
- It is often desirable to execute some consecutive goals inside an ECGE as a sequential unit. This can be done by bracketing the goals into one and-goal. Each such and-goal is known as an and-block in DDAS. An example is shown in Figure 12.

  RAP-WAM's instruction set is not able to deal with and-blocks directly, as the units of parallel execution are simple Prolog goals. A source-to-source transformation is needed to compile this type of code by "wrapping" the goals in an and-block into new clauses, which are called from inside the ECGE. This can be done transparently during compilation, but does impose a slight overhead in adding an extra level of procedure call. In DASWAM, and-blocks are supported directly: if the and-block consists of more than one goal, then a par_proceed instruction is added after the last call in the and-block only. This will have the effect of invoking the and-parallel scheduler only at the end of an and-block.

The format of the par_proceed instruction is: E is the address for the code that is to be executed when the ECGE succeeds, and s is the slot number of the and-goal that is being completed.

```
traverse_tree(tree(Left,Right,Node), LData, RData) :-
        mark_node(Node) &
        (traverse_tree(Left, LData1, RData1),
         process_data(LData1, RData1, LData))
        &
        (traverse_tree(Right, LData2, RData2),
         process_data(LData2, RData2, RData)).
```

FIGURE 12. Example clause with and-blocks of more than one goal.

5.1.1. Dependent Annotation Instructions. The following instructions are used to annotate a dependent variable/term as dependent at the start of an ECGE. Additional instructions are used to implement the ground and independence tests that are needed for both IAP and DAP. These tests are the same as those used for RAP-WAM (except that suspensions may occur during the test), so they will not be discussed here.

make_x_dependent(A_i, j)
This instruction dereferences and converts the term in the argument register A_i to a dependent term. The first occurrence of the dependent variable is in and-block j. Suspension occurs if the term contains dependent variables that are in a consumer position for the current task.

make_y_dependent(Y_i, j)
Similar to make_x_dependent. Converts the term in Y_i to a dependent term.

5.2. Modifications to Existing Instructions

Although the number of extra instructions is small, many sequential WAM instructions had to be modified to support DAP. Most important of these modifications is to allow the unification routines to support the suspension and resumption of execution of a goal. Also, the problem of synchronizing the creation and consumption of dependent compound terms, which takes place over more than one DASWAM instruction, are dealt with by writing special "invalid" values to the arguments that have not yet been initialized when the compound term is created. This approach has the advantage that it does not require changes to the code that a sequential WAM compiler generates in these cases.

However, there are some cases where goals in an ECGE have to be compiled somewhat differently from sequential goals: if a variable occurs in more than one goal, then the sequential Prolog compilation assumes that the variable is initialized by the leftmost goal it occurs in, and the code for the other goals assumes that the said variable is initialized. In DDAS, this assumption may not be valid, and such variables have to be initialized before the and-parallel execution is started. This can be done mostly by the use of existing WAM instructions, but two extra instructions are needed to deal with nonstrictly independent variables which occur inside structures. Such variables have to be globalized before the ECGE is entered. Since the need for these instructions is not directly related to DAP, they will not be discussed further. The interested reader is referred to [27] for a fuller discussion of the reasons for these instructions:

put_y_local_variable(Y_n, X_i)
put_y_local_value(Y_n, X_i)

5.3. An Example Compiled Code

In this section, an example of the compiled DASWAM code shall be shown, in order to illustrate how the instructions are actually used in practice. The codes shown are essentially what the current DASWAM compiler produced, except that the code

```
traverse_tree/3/1:
   get_structure_x0(tree/3)    % traverse_tree(tree(Left,Right,Node),
   allocate                    %                LData,RData) :-
   get_y_variable(Y6,A1)
   get_y_variable(Y2,A2)
   unify_y_variable(Y7)
   unify_y_variable(Y3)
   unify_x_variable(A0)
   put_y_void(Y0)
   put_y_void(Y1)
   put_y_void(Y4)
   put_y_void(Y5)
   allocate2
   allocate_pcall(3,-1,8)      % (
   push_goal(L1, 1, -1, -1)    % -1 as blocks are independent, so no
   push_goal(L2, 2, -1, -1)    % left/right siblings
   call(mark_node/1,8)         % mark_node(Node)
   par_proceed(L3, 0)          % &
L1:                            % (
   put_y_value(Y7,A0)
   put_y_value(Y4,A1)
   put_y_value(Y5,A2)
   call(traverse_tree/3,7)     % traverse_tree(Left,LData1,RData1),
   put_y_unsafe_value(Y4,A0)
   put_y_unsafe_value(Y5,A1)
   put_y_value(Y6,A2)
   call(process_data/3,4)      % process_data(LData1,RData1,LData)
   par_proceed(L3,1)           % ) & (only at end of an and-block)
L2:                            % (
   put_y_value(Y3,A0)
   put_y_value(Y0,A1)
   put_y_value(Y1,A2)
   call(traverse_tree/3,3)     % traverse_tree(Right,LData2,RData2),
   put_y_unsafe_value(Y0,A0)
   put_y_unsafe_value(Y1,A1)
   put_y_value(Y2,A2)
   call(process_data/3,3)      % process_data(LData2,RData2,RData)
   par_proceed(L3,2)           % ).
L3:
   deallocate
   proceed
```

FIGURE 13. Compiled DASWAM code implementing and-blocks with more than one goal.

generated by the compiler still treats the whole ECGE as one group, and the code had to be modified to take advantage of groups.

The codes have also been somewhat simplified (last call optimization has been removed), and edited to make them more readable. Note the notation used: `An` for the argument register n, `Yn` for environment variable n, `ns` for slot n (in the parcall frame), `nl` to indicate the left and-goal in the same group is n, `nr` to indicate the right and-goal in the same group is n. -1 for n represents the null value.

Figure 13 shows the compiled code of a program fragment with an ECGE of three groups of independent and-blocks, as shown in Figure 12. Two of the and-blocks have more than one goal. As discussed, this code can be compiled directly into DASWAM instructions.

6. DEALING WITH NONPURE FEATURES

One of the design aims of DDAS is that it should be able to handle full Prolog. Thus, the impure features of Prolog must be handled by DASWAM. The nonpure features can be generally classified into two types: metalogical and extralogical. These are discussed separately. In addition, cut, an extra-logical feature, is discussed separately from the other extra-logical features, both because of its importance, and also because of the special way in which it is handled.[18]

6.1. Metalogical Features

The dynamic producer feature of DDAS is sufficient to handle metalogical predicates such as `var/1`. The value of an unbound dependent variable accessed from a consumer position should be regarded as "unknown" rather than "unbound." Thus, when a metalogical predicate tries to find the state of an "unbound" dependent variable from a consumer position, the predicate must suspend until the value of the variable is known, either because it becomes instantiated by the producer, or the suspended task becomes the producer. Thus, all that is needed to handle metalogical predicates is to code such predicates so that they are able to use the normal DASWAM suspension mechanism to handle dependent variables.

It is also possible to provide a limited form of "nonstrict" DAP using special forms of the metalogical predicates that do not suspend on dependent variables in a consumer position. "Nonstrict" DAP occurs when it is possible to allow a consumer to nv-bind or access the state of a dependent variable in a consumer position without breaking the equivalence to sequential Prolog. This can occur, for example, when it is known that the current producer for the dependent variable will no longer generate bindings for the dependent variable, and thus it is safe for the next leftmost goal sharing the variable to become its producer. Obviously, "nonstrict" DAP can allow more parallelism to be exploited, and its exploitation and implementation is an area of further research. One complication in its implementation is that the producer status has to be passed from producer to a consumer sometime during the execution of a goal, instead of at the end of its execution. However, if the non-suspending metalogical predicate is used correctly, i.e., it is only used to test for the state of a dependent variable in a consumer position where it is known that the current producer would not change the state, then it can lead to the exploitation of "nonstrict" DAP in these situations without the need to develop special support for "nonstrict" DAP. Of course, if used incorrectly, it can lead to inconsistent parallel executions.

6.2. Pruning Operations: Cut

Like &-Prolog, and unlike the committed choice languages and Andorra, goals to the left and right of the pruning operator (cut for DDAS) can execute in parallel at the same time. However, the pruning needs to be coordinated across these goals, when the cut prunes away search-space in other and-goals.

In the SICStus WAM, when a cut is encountered, the current choice-point register is set to point at the last choice-point that is outside the scope of the cut. However,

[18]The general method for handling extralogical features can certainly be applied to cuts as well, but it would unnecessarily restrict parallelism.

such a simple scheme is not sufficient for a distributed stack, as the choice-point to reset to may be in a different and-section.

Three general situations can be recognized when a cut is encountered in DASWAM:

- The cut cuts to a choice-point within the current section. The normal SICStus WAM cut mechanism is used to deal with this.
- The cut cuts across sibling and-goals to its left. An example of this is:

  ```
  foo :- (true -> a & b & (c, !)  & d).
  ```

  This cut cuts away the choices of `a`, `b`, `c`, and `foo`. The main problem is that `a` and `b` are executing in parallel, and may still be executing when the cut is encountered. The effect of the cut is thus performed in two stages: the choices of `c` are pruned when the cut is encountered. The slot associated with `a` and `b` is then marked with a "cut" flag. The pruning of choices on `a` and `b` then takes place when all sibling and-goals between them and the cut have returned a solution, i.e., `b` is pruned when `b` returns a solution, `a` is pruned when both `a` and `b` have returned a solution.
- The cut cuts to a choice-point outside the current section, but does not cut across sibling and-goals. In this case, the pruning can take place immediately.

In order to perform the pruning, the worker encountering the cut must follow the markers backward up the stack sections, setting the pointers in the markers to discard the choice-points in the sections, until the choice-point/marker to cut to is located. In the case of the cutting across sibling and-goals, the first stage cuts to the marker representing the start of the current task, i.e., `c` in the above example. In the second stage, `b` is pruned to the marker representing the start of the task executing `b`, and for `a`, the pruning carries on untill the choice-point for the parent goal (`foo` in the example) is pruned. The reason that `a` cannot be pruned before `b` is finished is because `b` might fail due to some dependent binding it consumed, and this would cause `a` to generate another solution before the cut would be encountered in sequential execution (after `c`).

In the current DASWAM, the space represented by the discarded choice-points cannot be immediately recovered, as the markers have to be retained to allow untrailing of variables during the actual backtracking. The space can be recovered by a garbage collector, or alternatively, if the control stack is separated into a separate choice-point and marker stacks. Some redesign of the existing scheme would be needed, but in principle this would make the recovery of the space occupied by the choice-points easier.

Note that no parallelism is lost (except for whatever overhead is needed to perform the cut) in dealing with cuts. This is in contrast to dealing with other side effects, where the task performing the side effect must in general suspend until it is leftmost.

6.3. Extralogical Features

Extralogical predicates include those predicates that handle I/O, and predicates such as assert/retract that affect the global state of the computation. All these predicates need some synchronization with the execution of other parts of the program, but to different degrees. Here are the two possibilities:

DDAS style synchronization: This is the synchronization whereby a predicate is allowed to proceed when it is in the producer position. This synchronization is sufficient to ensure that output would produce the same output as the sequential case. It may also be sufficient for certain (nonstandard) restricted use of assert/retract-type predicates where the "scope" of the assert/retract is limited. The existing machinery for synchronization in DASWAM is sufficient to support this form of synchronization.

Global synchronization: This form of synchronization ensures that the events being synchronized in the program occur exactly in the same order as they would sequentially. It may, for example, be desirable for the output to appear in the same order as it does sequentially, which the DDAS style synchronization does not ensure. Inputs from a sequential access source such as a file also need this form of synchronization, as do the predicates that modify the global state.

Extra machinery has to be added to DASWAM to support this form of synchronization, but methods developed to support such forms of synchronization for IAP, such as that proposed by Muthukumar and Hermenegildo [17], can be used directly in DASWAM as well, because the addition of DAP does not affect this form of synchronization.

7. RESULTS

Results for DASWAM running benchmark-type programs have been presented previously [27, 28]. Such programs are useful for showing that DASWAM can indeed be implemented, and can extract some parallelism, but they do not really show that DASWAM can extract effective parallelism from realistic programs. As the prototype DASWAM is now stable enough to run reasonably sized application programs, results for such programs will be presented here. However, as application programs are much less readily available than benchmark-type programs, so far we have been able to study only a few applications. Here, two application programs are presented in some detail: the British Telecom Workforce Management program (wms), and the author's own or-parallel simulator [30, 31], Orsim. These two programs are known to have some and-parallelism: DAP in wms, and IAP in Orsim. Thus, the aim is not to show that DDAS/DASWAM can extract significant parallelism from general application programs—that would need a much more extensive study than two programs—but to show that if an application program does contain and-parallelism (both IAP and DAP), DDAS/DASWAM is able to effectively extract such parallelism.

In addition, results for several benchmark-type programs are presented in summary form. However, space constraints prevent the detailed presentation of results for these programs.

7.1. Description of Programs

7.1.1. Benchmark-Type Programs

Boyer: This is the standard "boyer" benchmark [34]. This program exploits (nonstrict) IAP only.

Bt_cluster: This is the clustering benchmark from British Telecom Research Laboratory. 500 elements are being clustered in this example. This program exploits (nonstrict) IAP only.

Nrev: Naive reverse of a 400-element list. This program exploits DAP.

Qsort: Quick-sort of a 1600-element list. The list consists of a list of 400 randomly generated integers repeated four times. This program exploits both DAP and IAP.

Kkqueens: This is the kkqueens benchmark by K. Kumon for committed choice languages. It solves the N-Queens problem (N = 9 here). The code used here is taken from the benchmark set that comes with the sequential version of KL1 distributed by ICOT [4], and slightly modified so that it would run as a Prolog program. Two annotations for this program were tried, both exploiting DAP: in DAP1, one clause was annotated with an ECGE; in DAP2, an additional clause was annotated with an ECGE.

7.1.2. Or-Parallel Simulator. The or-parallel simulator was developed as a Prolog program to study the behavior of or-parallelism [25, 31]. It was used in a subsequent study of or-parallelism and IAP [27, 30].

The parallelism was extracted by just one CGE, with a very minor rewrite of one clause of the program to remove some unnecessary dependencies between goals that could be executed in and-parallel independently. DDAS was able to exploit this IAP exactly as a system that only exploits IAP, such as &-Prolog.

The program consists of 1071 lines of code. The example studied was the simulator simulating the execution of the standard population density benchmark "query."

7.1.3. Workforce Management Program (WMS). wms is a series of programs developed by D. Munaf at British Telecom Research Laboratory [16]. These programs are designed to solve the problem of allocating a set of jobs effectively to a set of engineers, based at several different locations. A series of programs were developed to improve the execution characteristics on Andorra-I. There are three main stages to the development:

wms0: The original program had very little potential for parallelism because of the algorithm used.

wms1: wms0 was completely rewritten using a different algorithm that was intended to offer more parallelism. However, Andorra-I was not able to extract much parallelism from this program. This program gave a better solution (significantly more jobs were allocated) than wms0.

wms2: wms2 was extensively modified (mainly by the Andorra-I group) so that Andorra-I was able to extract parallelism from it. This involved rewriting many parts of the program deterministically. In particular, some predicates that contain clauses which have the equivalent of "deep guards"—clauses which contain user-defined goals designed to eliminate all but one of the clauses of the predicate—had to be rewritten.

In addition, two further changes were needed: (1) a `var/1` in one goal prevented significant and-parallelism: this is because the standard `var/1` of Andorra-I suspends until it is leftmost to preserve Prolog semantics [23]. This had to be replaced by a nonsuspending `var/1` and the code modified

accordingly to allow the nonsuspending `var/1` to be used correctly. Such a `var/1` is similar to the "nonstrict" `var/1` of DASWAM.[19] (2) The order of arguments passed to a predicate that appended two lists had to be reversed.

wms2 uses the same algorithm as that of wms1, and returned the same answers.

For DDAS, both wms1 and wms2 were parallelized with minor modifications and annotations. The modifications needed for wms1 were the use of "nonstrict" `var/1` and the change of code around it (essentially the delaying of the binding of the dependent variable to be tested by the "nonstrict" `var/1`), and the reversal of the arguments to append. The changes were localized to three clauses. Three different annotations were then used to evaluate their performance on DASWAM:

IAP: One clause (the only one expected to have IAP) of the program was annotated to extract IAP only.

DAP0: One clause of the program was annotated to extract DAP.

DAP1: One extra clause (the one annotated for IAP) to DAP0 was annotated to extract DAP. These two clauses were expected to be the major sources of parallelism.

DAP2: One extra clause to DAP1, which was not expected to give significant parallelism, was also annotated, giving three annotated clauses.

For parallelizing wms2, ordering of some goals was changed to make the clauses tail-recursive, and allow the dependent bindings to be passed between phases of the program quicker, thus increasing the amount of parallelism. The two equivalent ECGEs to DAP1 above were added to extract the parallelism.

The example problem is to try and allocate 250 jobs to 118 engineers from 9 bases. wms1 consists of 1912 lines of code, of which about 1280 lines are the database on the jobs and engineers. wms2 uses the same database, and is 2119 lines long.

7.2. Analysis of Parallel Execution Overheads

One important question is what overheads are imposed by running programs on DASWAM. The overheads can be broadly divided into the following:

1. Overheads of running an unannotated program on DASWAM, e.g., use of unbound tag.
2. Overheads associated with IAP on one worker, e.g., allocation of markers, execution of parallel instructions, some lockings.
3. Overhead associated with DAP specifically on one worker, e.g., checking of producer status, use of dependent cells, extra lockings on dependent cells. Note that a significant portion of the overheads of supporting dependent cells is to allow correct backward execution: all dependent bindings have to be trailed, and dependent cells created even for bound variables because their contents may be consumed and thus need to be killed during backtracking.
4. Parallelization overheads: exploiting IAP on more than one worker; e.g., coordination of workers, finding work, coordinating parallel backtracking, hardware dependent overheads (such as accessing shared memory).

[19] In fact, the nonsuspending `var/1` of Andorra-I inspired the "nonstrict" `var/1` of DASWAM.

5. Parallelization overheads: exploiting DAP on more than one worker; e.g., suspension and unsuspension of tasks, extra hardware dependent overheads due to greater sharing of memory.

The DASWAM prototype simulates parallelism, and therefore cannot directly measure the last two overheads. The speedups it gives are an indication of the potential parallelism that DDAS/DASWAM is able to extract from a particular program. It would then be necessary to combine these results with that from a parallel implementation to measure the parallel overheads.[20]

However, the prototype can throw some light on the first three types of overheads. In fact, it should incur all the overheads of executing IAP and DAP programs on one worker of a real parallel system except those due to lockings.[21] Table 1 shows the execution times of the programs: the column "WAM" is the times for executing the programs on the sequential SICStus WAM that DASWAM is based on,[22] "DASWAM" is the times for executing the unannotated programs on DASWAM, "IAP" is the times for executing the programs annotated for IAP on DASWAM with one worker, and "DAP" is the execution times for programs annotated for DAP; if more than one DAP annotations were tried, these are shown in separate rows. All the measurements are the average of three execution times (in ms) for each program (on a SparcStation 20). The standard deviations are included in the data.

Table 2 provides additional statistics gathered from running the parallelized programs. "Diff" is the slowdown of their execution times with respect to the

[20] A parallel implementation on its own cannot measure parallel overheads either, because the effects of available parallelism on speedups cannot then be separated from the overheads. See [30] for a more detailed discussion of this.

[21] This was confirmed by the new real parallel system: initial results show that the performance on one worker is similar to the simulator's performance; in fact, the performance may even be somewhat better than the simulator's.

[22] "WAM" is a sequential WAM developed by the author from the SICStus 0.6 specification. It runs somewhat slower than SICStus, with approximately the same speed as another widely available Prolog system from ECRC, Eclipse (with the programs compiled for no debugging to maximize speed).

TABLE 1. Execution times for programs on WAM and DASWAM.

Program	WAM	DASWAM	IAP	DAP
boyer	2840±22	3663±19	7097±22	—
bt_cluster	3647±5	4853±5	4870±14	—
nrev	353±9	447±12	—	1183±5
qsort	220±8	273±5	327±12	673±12
kkqueens	1420±8	1880±8	—	
(DAP1)				4903±12
(DAP2)				5837±39
Orsim	5313±12	7143±33	7340±8	—
wms1	1800±14	2387±33	2500±8	
(DAP0)				2893±45
(DAP1)				3150±24
(DAP2)				3540±22
wms2	2343±21	3276±12	3333±33	3900±22

TABLE 2. Execution statistics for parallelised programs.

Program	Diff	∑deref	∑prod. check	∑inst	∑ECGE	∑par	Σusp(10)	Par(10)
kkqueens, dap2	3.10×	1572972	482417 (30.7%)	2940969	32404	129616 (23)	21907 (134)	6.33×
nrev, dap	2.65×	401804	318803 (79.3%)	730621	400	1600 (457)	399 (1831)	9.15×
kkqueens, dap1	2.61×	1492573	350938 (23.5%)	2857039	24011	96044 (30)	22133 (129)	7.07×
qsort, dap	2.47×	178248	95399 (53.5%)	331610	1600	9600 (35)	10262 (32)	8.92×
boyer, iap	1.94×	1628574	0	5373160	94070	376280 (14)	0	9.80×
wms1, dap2	1.48×	1035845	185355 (17.9%)	3170854	2338	11390 (278)	12250 (259)	8.26×
wms1, dap1	1.32×	999605	130716 (13.1%)	3148894	508	4070 (774)	12632 (249)	8.02×
wms1, dap0	1.21×	988353	120335 (12.2%)	3136009	507	4056 (773)	17554 (179)	6.16×
qsort, iap	1.20×	104318	0	360639	1600	6400 (56)	0	3.27×
wms2, dap1	1.19×	1770897	110425 (6.24%)	4368113	508	4070 (1077)	12759 (342)	8.31×
wms1, iap	1.05×	886208	0	3137259	1	4 (784315)	0	1.09×
orsim, iap	1.03×	3011140	0	9069226	1200	4800 (1889)	0	9.79×
wms2, iap	1.02×	1687630	0	4358459	1	4 (1089615)	0	1.01×
bt_cluster, iap	1.00×	3125344	0	6044198	500	2000 (3020)	0	9.74×

execution time for running the unannotated program on DASWAM; thus it is a measure of the overheads due to the DAP and IAP on one worker—the programs are listed in the reversed order of their slowdown, so "kkqueens,dap2," which runs 3.10 times slower than the unannotated program, is listed first. The next column, "$\sum$deref" is the number of dereferencing operations performed when executing the program. "$\sum$prod. check" is the total number of checks for a dependent variable's producer/consumer status, with the bracketed % being with respect to $\sum$deref (the vast majority of the checks are performed during the dereferencing of a dependent variable, so this column shows the effectiveness of distinguishing dependent and nondependent variables: if all variables are treated as dependent, then the % would all be approximately 100%). "$\sum$inst" is the total number of DASWAM instructions executed by the programs. "$\sum$ECGE" is the number of ECGE allocated in the program, and $\sum$par. is the total number of "parallel instructions" (`allocate_pcall, push_goal, par_proceed`) executed, with the figure in brackets showing the average number of all instructions executed before a parallel instruction is executed. All the columns discussed so far are gathered with the program executing under one worker, but the last two columns of the table are for data gathered from the execution under 10 workers: $\sum$susp(10) shows the number of suspensions performed, the bracketed figures are the average number of DASWAM instructions (from $\sum$inst) executed before a suspension occurs. $\sum$Par(10) shows the "speedup" (or as discussed, inherent amount of parallelism) relative to one worker.

Table 1 shows that DASWAM is able to run an unannotated program at between 71 to 81% of the speed at which the program runs on a WAM. The overhead here is probably due mainly to the use of unbound tags (instead of self-referencing pointers) to represent unbound variables, and to the trailing of two words per trailed item (instead of one). This overhead is consistent with the cost reported for a similar unbound tag mechanism in Aurora [32]. For DASWAM, the unbound tag is used to preserve the "two-stack model" of the WAM by allowing comparison of seniority of variables in different stack sections in the local stacks (see [27] for details), thus avoiding the globalization of many local stack variables. As shall be discussed shortly, it is not clear if the benefits of preserving the two-stack model justify the overhead, and the alternative scheme of globalizing most variables might be a better option.

The IAP and DAP overheads on one worker for DASWAM include most of the parallel overheads, except those associated with suspension and some of the cost in coordination of workers, but these do not apply in the one-worker case. Table 2 provides some data to help in analyzing these overheads. The data suggest that exploiting DAP does impose more overhead than exploiting IAP, although an important parallel overhead is due to the execution of the parallel instructions. The data suggest that the execution of a parallel instruction is significantly slower than that of the normal instructions; currently, the parallel instructions are coded quite naively, and it is expected that with some effort, the time needed to execute these instructions can be reduced significantly, thus leading to much less slowdown of the parallelized programs. However, even with the current implementation, the data show that the overheads are much more significant for the simple benchmark type programs, which tend to execute far fewer instructions per parallel instructions than the more realistic application programs: the parallelized application programs run

at between 67 to 97% of the speed of unannotated program on DASWAM, and 51 to 73% the speed on the sequential WAM.

The use of the dependent variable imposes extra overheads on DAP programs. The data ($\sum$prod. check) shows the benefit of DDAS's distinguishing of dependent and nondependent variables: only 6 to 18% of references for applications, and 23 to 80% for benchmarks, needed the checking of producer status. Dependent variables are accessed in many instructions, but the data suggest that the execution of parallel instructions still dominates the DAP overheads: the ordering of programs by their amount of slowdown from the annotated program (as is done in Table 2) corresponds very well with ordering the programs by the number of instructions executed per parallel instruction (ordering by the bracketed figures in $\sum$par., smallest first); the main exception is nrev, which executes a relatively large 457 instructions per parallel instruction, but has the second greatest slowdown for DAP programs. One reason for this might be that the program also has the highest proportion of checking for producer status (79.3%) by a significant margin.

The ability to control the amount of annotation, a feature of DDAS, is important: as expected, the data show that different annotations for the same program lead to different parallelized behavior: generally the more annotations, the greater the slowdown of the program running on one worker—there are a greater proportion of parallel instructions and producer status checks. Thus, unless the extra annotation leads to greater speedups that compensate for the slower execution, it is undesirable to have these extra annotations. The prototype is unable to directly measure real parallel speedups, but the predicted parallelism can serve as an upper bound on the real speedups. Thus, for wms, IAP is unable to extract much parallelism at all; DAP0 is able to extract most of the parallelism, at a relatively low cost; DAP1 is able to extract somewhat more parallelism, at a higher cost, but the extra parallelism may be able to compensate for this; DAP2 is able to extract only slightly more parallelism than DAP1, but at quite a significantly higher cost, and the slight extra parallelism is unable to compensate for this. For kkqueens, the extra annotations of DAP2 impose extra cost without an increase in parallelism (in fact, for 10 workers, the parallelism for DAP2 is *lower* than that for DAP1). For qsort, the DAP version imposes much higher overheads than the IAP version, and although the parallelism is also significantly more, it is not clear if it is sufficient to compensate for the extra cost, as it is expected that achieving DAP speedups would be more expensive than the IAP speedups.[23]

Another potential problem with exploiting DAP in DASWAM is the cost of determining the producer/consumer status: it is potentially unbounded—when an unbound dependent variable is accessed in an ECGE that is separated by arbitrarily many levels of intervening ECGEs from the ECGE in which the variable became dependent (see Section 4.2.3), all the intervening ECGEs have to be traversed. Table 3 summarizes the ECGE traversal data for the DAP programs in our study. Each column shows the % of finding current task position that traversed the number of ECGEs indicated by the column; so, for example, "1" represents the % of tests

[23]The IAP speedups achieved by qsort for the example list may be atypically high, because the list consists of repeating a 400-element list three times; being able to divide the existing list into roughly equally sized sublists is important to achieving good speedups under IAP. Thus, in a completely randomly generated list of the same size, it is expected that the difference in parallelism would be more favorable to the DAP program.

TABLE 3. Data for checking producer status.

Program	Levels of ECGE traversed (%)												
	0	1	2	3	4	5	6	7	8	9	10	>10	Max
nrev.	75.0	25.0	0	0	0	0	0	0	0	0	0	0	1 (730621)
qsort	29.3	70.7	0	0	0	0	0	0	0	0	0	0	1 (67460)
kkqueens, dap1	31.3	8.62	9.10	7.79	6.22	4.98	4.25	4.14	3.64	3.33	2.77	13.9	27 (23)
kkqueens, dap2	25.4	13.3	10.2	10.3	6.37	4.73	3.13	2.75	2.28	2.50	2.21	16.9	34 (1)
wms1, dap0	54.8	45.2	0	0	0	0	0	0	0	0	0	0	1 (54450)
wms1, dap1	52.3	41.9	0.187	0.172	0.154	0.137	0.195	0.138	0.184	0.199	0.138	4.25	118 (2)
wms1, dap2	47.9	34.7	2.62	2.00	1.18	1.08	1.05	0.863	0.801	0.681	0.538	6.59	118 (2)
wms2, dap1	61.4	31.9	0.212	0.195	0.172	0.152	0.211	0.158	0.209	0.226	0.154	4.97	118 (2)

that traversed 1 ECGE. ">10" is the % of checks that had to traverse more than 10 ECGEs. The last column, "max," shows the maximum number of ECGEs traversed in any check, and the number of times it occurred in brackets.

All the programs were executed with one worker. As the checking is performed regardless of how many workers are running the program, and the number of levels traversed does not change with number of workers, so the results presented are essentially independent of the number of workers.

The results show that the majority of checks traverse only a few ECGEs for all the programs, although there are a few checks that do traverse many levels of ECGE. However, such checks are relatively infrequent, and the results from Tables 1 and 2 suggest that they do not have a large impact on the performance: both nrev and qsort did not need to traverse more than 1 ECGE for all checks, but they have greater slowdown than most programs which have to perform more traversals in their checks—nrev has the second greatest slowdown of all programs. For wms1, the DAP0 version shows that just removing one ECGE can have a very large impact on the traversal pattern: DAP1 has only a very slightly higher proportion of parallel instruction, a somewhat higher number of checking for producer, but has a long tail of traversing large numbers of ECGEs during the checks; nevertheless, DAP0 executes only slightly faster than DAP1, and it is not clear if this is due mainly to the ECGE traversal, or the higher number of producer checks. In any case, the impact of the dramatic change in traversal pattern is minimal. Taken together, all these data suggest that the number of ECGE traversed for producer/consumer checking is not a significant factor affecting performance.

The last two columns of Table 2 show some data from running the programs with 10 workers. The data show that DDAS is able to extract effective parallelism over a large range of programs, including those with determinate DAP (exploited by committed choice languages and Andorra-I), e.g., kkqueens, wms2; those with nondeterminate IAP (exploited by &-Prolog), e.g., Orsim; and those with nondeterminate DAP (exploited by DDAS only), e.g., wms1. Although the prototype simulator cannot really throw much light on the actual real speedups on its own, $\sum$susp does give some indication of one source of parallel dependent overhead: that of the number of suspensions performed. This ranges from 32 instructions per suspension for qsort, to 1831 instructions per suspension for nrev; these results suggest that it may be more difficult to obtain good speedups for qsort than nrev, if the cost of suspension is an important source of parallel overhead. Indeed, this has been confirmed by the initial results from the parallel implementation.

The design aim of the prototype was to verify the concepts of DDAS/DASWAM, and there were very few optimization efforts, so there should be ample opportunities to reduce the overheads. The overheads with respect to the WAM includes the overheads for the use of the unbound tag, which the data suggests may be a more significant overhead than either the DAP or IAP overheads for the applications programs. This overhead can be eliminated if an alternative scheme (e.g., globalization of most local variables) is used, and thus the overheads with respect to unannotated programs running on DASWAM may be a better representation of the IAP and DAP overheads. Tuning of the code in the prototype should lead to lowering of the overheads, and in addition, compile-time analysis has the potential to greatly reduce the DAP overheads, as a significant portion of the DAP overheads might be avoidable if it can be determined by analysis that the dependent binding would not be undone by inside backtracking and thus no

support for complex backtracking, currently provided for every dependent variable, is needed. Thus, the potential exists for very significant reduction in the overheads, but even currently, the overheads are significant but are certainly not prohibitively expensive.

7.3. Parallelism Extracted from Application Programs

Figure 14 shows the speedups for Orsim and various wms obtained from DASWAM. The speedups are relative to each program running on one worker. For both programs, the speedups obtained from the new prototype parallel DASWAM on a

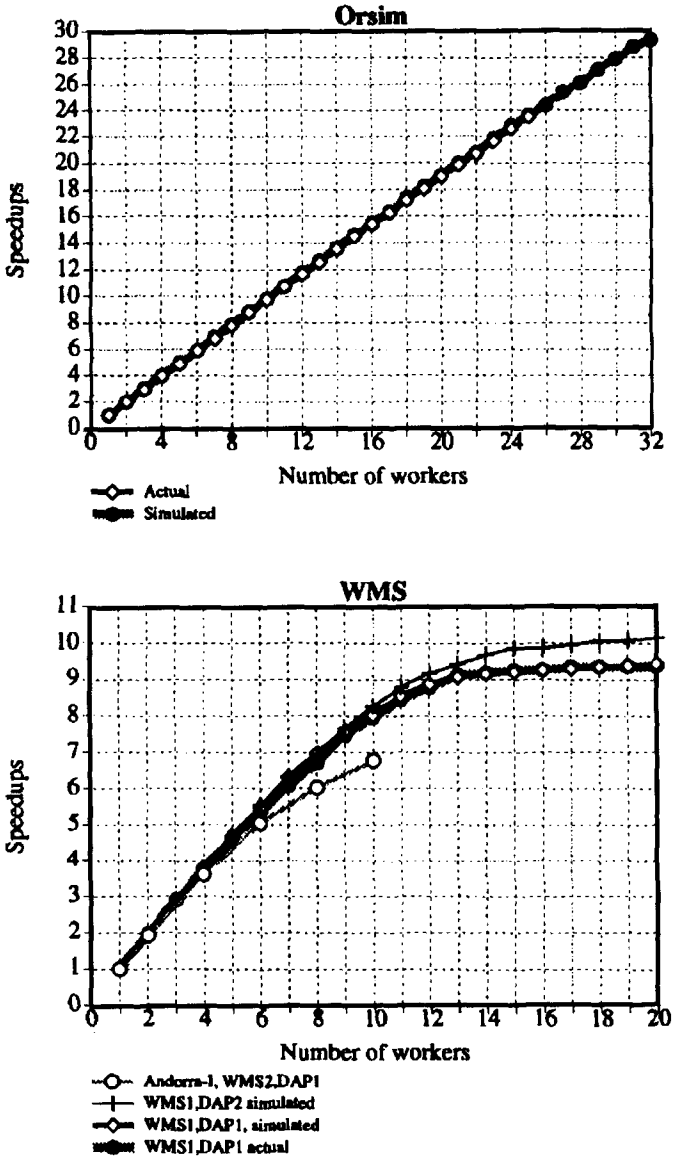

FIGURE 14. Parallelism extracted by DASWAM for Orsim and wms.

26-processor Sequent Symmetry are included for reference. For wms, speedups from the simulator are shown for wms1 with IAP, DAP1, and DAP2 annotations, along with real speedups for DAP1; speedups obtained for Andorra-I running wms2 on a Sequent Symmetry are also included for reference. The speedups for wms2 with DAP1 annotations and DAP0 for wms1 were not included to avoid cluttering the graph—it is very similar to that of wms1 DAP1.

Orsim contains significant IAP. Wms contains significant DAP, although the speedup begins to flatten at around 12 workers. However, as the size of the example problem used is smaller than the real problems, and the parallelism generally increases with problem size in this particular program, there should be more parallelism in real problems. Both DASWAM and Andorra-I are able to extract significant DAP from wms, although DASWAM seems able to give slightly better speedups.

The new parallel results show that DASWAM is able to effectively exploit the parallelism extracted by DDAS—the speedups from the new parallel implementation, although still very preliminary, are very close to the predicted speedups from the simulator. These results strongly suggest that the DASWAM is able to produce the predicted speedups in Table 2, and that parallel execution would produce much better performance than a sequential Prolog system can.

7.4. Memory Usages

Table 4 shows the memory usage (in words) of Orsim and wms running under DASWAM with 1 and 10 workers. The data were gathered at one particular

TABLE 4. Memory usage of orsim and wms under DASWAM.

Program	Local	Control	Global	Trail	Dtrail	Dheap	Goal	Total
orsim (WAM)	3683	1608	600832	93319	—	—	—	699442 (1×)
orsim iap	45965	89366	601548	131370	0	0	2400	870649 (1.2×)
orsim iap, 10w	45983	88551	600672	130016	0	0	0	867797 (1.2×)
max	4955	9400	62551	14042	0	0	0	89050
min	4277	8792	58133	11384	0	0	0	84720
sd	±216	±299	±1288	±748	±0	±0	±0	±1366
wms1 (WAM)	38	48	272630	3965	—	—	—	276681 (1×)
wms1, iap	160	67	271458	5276	0	0	2	276411 (1.0×)
wms1, dap1	10742	44755	279799	11772	152668	217140	3054	719930 (2.6×)
wms1, dap2	34470	143586	303168	4510	242086	321136	6714	1055670 (3.8×)
wms1, dap1, 10w	14038	525821	394359	15774	145512	217660	0	1313164 (4.7×)
max	1781	57100	41546	1916	16084	25852	0	142191
min	1125	45384	35612	1350	13012	19416	0	121216
sd	±181	±4122	±2002	±180	±1023	±1820	±0	±7580
wms1, dap2, 10w	35009	616527	412913	8924	220356	321676	0	1615405 (5.8×)
max	3799	64981	45139	1064	23058	33276	0	169732
min	2960	58115	38663	746	19784	29332	0	153225
sd	±225	±3160	±2394	±101	±1370	±2056	±0	±7953
wms2 (WAM)	1117	0	548853	2393	—	—	—	552363 (2.0×)
wms2, dap1	10769	44755	560657	12646	152738	192788	3054	977407 (3.5×)
wms2, dap1, 10w	38202	545096	678795	16182	144488	193288	0	1616051 (5.8×)
max	4597	61066	72411	1938	17372	24716	0	187070
min	2392	39192	65412	1294	11636	14984	0	135314
sd	±593	±7434	±2283	±192	±1489	±2378	±0	±13005

instance of time near the end of the execution of each of the above programs, where the memory usages are very near their maximum,[24] and thus the various versions of the programs can be compared for memory efficiency. The data presented for the 10 worker cases each consist of four rows: the first row shows the total combined memory usage of all 10 workers for the particular memory area in each column; the second row (max) shows the maximum usage out of the 10 workers for the particular memory area; the third row (min) shows the minimum usage out of the 10 workers; and sd shows the standard deviation on the average memory usages of the 10 workers, giving some indication of the amount of variation in the particular memory area. The memory areas represented by the columns are: local—local stack; control—control stack; global—global stack; trail—trail; dtrail—dependent trail; dheap—dependent heap; goal—goal stack; total—total usage of all stacks; the figure in brackets shows the memory usage relative to the sequential execution. In the case of the wms programs, they are relative to the wms1 usages.

It is beyond the scope of this paper to discuss the memory usage in great detail, so only a short discussion of the most important observations will be presented here.

7.4.1. IAP Overheads

- IAP does not impose a very heavy overhead on memory usages: wms1, IAP uses essentially the same amount of memory as wms1 on the WAM, and Orsim, IAP uses only about 20% more memory. For wms1, this is partly due to the lack of IAP, so it also shows that where there is little parallelism, sequential parts of the program behave much as they would on a WAM, memory-wise. Orsim, on the other hand, does contain significant parallelism, and this suggests that the IAP overheads are low for *application* programs.
- There is essentially no increase in total memory usage for Orsim running on 10 workers from the 1 worker case. This confirms that an annotated program running on one worker incurs almost all the parallel overheads except those associated with suspension and some of the coordination overheads. As there are no suspensions with IAP, and the coordination overheads do not affect memory usages, then essentially all the memory overheads are incurred even with one worker. With more workers, the memory usages stay more or less constant, and as long as there is sufficient parallelism, this memory usage is divided fairly evenly between all the workers.
- If only IAP is exploited, this incurs essentially no DAP overheads: the various memory areas associated specifically with dependent variables are empty in all the IAP executions.

7.4.2. DAP Overheads

- The executions of the various DAP versions on one worker all use significantly more memory than the sequential execution. Again, a significant amount of the overheads associated with DAP are paid for even with one worker. The dependent cells are allocated and trailed; the producer status is

[24]This is the case for these programs because, although they do perform some backtracking, these programs essentially produce one answer without much search. Thus the memory usage is the greatest just before the end of execution.

tested for, even though in the special case of one worker, dependent variables will always be accessed from a producer position. There is scope for optimizing the cases where sequentially consecutive tasks are executed sequentially and consecutively on the same worker: this would happen quite often when the number of available workers is small compared to the available paral lelism.

- In addition to the global stack, the memory usages of the dependent trail and dependent heap are the most significant in the one-worker case for the various versions of wms with DAP. Currently, the dependent trail also trails two words per item as does the normal trail, but there is little reason for this, as dependent cells are not allocated on the local stack and are in effect always globalized. Thus, only one work per item need be trailed, and this should halve the size of the dependent heap.

  It should be possible to further reduce the memory usage in both the dependent heap and dependent trail. For example, currently, all bindings to dependent variables are trailed. Recall that untrailing of dependent variables may trigger special actions; however, there should be no need to trail every single dependent binding, as the required action may have already been triggered by the detailing of another (later) dependent binding. In fact, such "redundant" trailings increase the complexity of the backward execution. It would be better for both forward and backward execution to trail only the first dependent binding in such cases.

  With the large memory usage of various memory areas with DAP, it is even less clear if retaining the two-stack model is useful or not: it probably does not save that much global stack usage, and in fact it may result in more overall memory usage because the trail size is increased.

- Unlike IAP, there is an increase in memory usage for DAP when the program is run with more than one worker. This is due mainly to the increased usage of the control stack, and most of this is due to the allocation of suspend and continuation markers. Currently, the size of a suspend marker is very large—22 words plus spaces for all the temporary and argument registers. There should be scope for reducing the size of the marker. Also, once a task is unsuspended and continued in another stack section, much of the storage of the suspended state of a task becomes garbage and can be garbage collected. This is not done currently.

  The dependent trail and dependent heap usages do not show any significant increases with more workers, as expected—as all the dependent cells are allocated even in the one-worker case. There is an increase in the global stack because the task suspend and variable suspend lists (see Section 4.2.2) are allocated on the global stacks.

- The memory usages for the DAP versions of the wms are fairly evenly distributed throughout the 10 workers, again because of the availability of parallelism. In addition, although the total memory usages in both the 1- and 10-worker cases are significantly more than in the sequential case, and can and should certainly be reduced, they are probably not too unreasonable when considered against the speedups. It is important to realize that if a parallel Prolog (and indeed any other parallel application) is to be used for practical purposes, it would be in a multiuser environment. In such an environment, it

is important to consider both processor time and memory as resources that are shared with other users. Thus, both the amount of resource used and the time the resource is actually used are important. Therefore, it is possible to trade amount of memory used against execution time. In such a scenario, using 4 to 6 times as much memory as the sequential case to obtain a speedup of 8 is acceptable.

Note also that the total memory usage can be reduced significantly by a garbage collector. However, because DASWAM is able to make use of stack memory management during backtracking in all memory areas except the dependent heap, the need for a garbage collector is not as great as in parallel logic programming systems that are implemented using just heaps rather than stacks.

- wms1 DAP2 uses significantly more memory than wms1 DAP1, but it is only able to extract a little extra parallelism. This again confirms one of the original assumptions of allowing annotations in DDAS: it can be counterproductive to allow all goals to run in parallel.
- wms2 uses more memory than wms1, both in the sequential and parallel versions. This is at least partly due to the nonfailing style of programming used: without failure and backtracking, space that is recovered in wms1 through backtracking is not recovered in wms2. It may be possible to recover such spaces if a garbage collector is available, but it is still better to recover such garbage through backtracking because the cost to do so would be smaller, especially considering the complexities of a concurrent garbage collector.

8. GENERAL DISCUSSIONS AND RELATED WORK

The experience in parallelizing the example programs and the results suggest that the design decision of allowing the user to indicate where parallelism should be exploited through ECGE is justified. Consider Orsim, which contains very complex clauses that are probably very difficult to analyze; it is perhaps impossible to extract the parallelism automatically. Certainly, neither the existing compile-time analysis tools of &-Prolog or Andorra-I were able to extract the parallelism. This does not mean that there is no need for automatic analysis—it would certainly be very useful, but the user should be allowed to supplement such automatic parallelization tools with user annotations. The important criterion is that such annotations should not add significantly to the programming complexity. As the ECGE annotations indicate where parallelism is to be exploited, they do not add significantly to the complexity, and this is confirmed by the ease of annotating the rather complex application programs presented here: as an annotator for DDAS does not yet exist, all the annotations were performed manually, without great difficulties.

One advantage DDAS has over systems that exploit deterministic DAP (e.g., Parallel Nu-Prolog, Andorra-I [19, 22]) is that it is able to exploit and-parallelism in nondeterministic goals. It is certainly possible to come up with simple programs which are not completely artificial that would benefit from such parallelism (e.g., the speaker example). It is, however, far from clear how useful such parallelism would be in real application programs. The real advantage of allowing nondeterministic

and-parallelism is that it may be easier to parallelize programs which contain deep guards. Such programs contain goals which return only one solution, but do contain nontrivial nondeterminacy in which all but one of the alternatives eventually fails. One example is wms, where considerable effort was needed to modify wms1 into wms2, so that the parallelizm that does exist in the algorithm can be extracted by Andorra-I, but DDAS was able to extract the parallelism from both programs.

It is, of course, possible to write programs from scratch in a style that is suitable for a system such as Andorra-I, but arguably such a style is less "natural" in Prolog, with few existing programs using such a style. The comparison between wms1 and wms2 certainly suggests that in some cases at least, the resulting program may be somewhat less efficient: wms2 runs about 30% slower and uses more memory, even on a mature Prolog system such as SICStus Prolog version 2.1.[25]

There have been some proposals to exploit "optimistic and-parallelism" [8, 21, 33] which may also be able to exploit nondeterminate DAP. However, as far as I know, none of these proposals has yet been seriously implemented. In addition, optimistic and-parallelism is inherently more speculative than DDAS, and requires a more complex backtracking scheme to maintain the equivalence to Prolog; it is thus likely that implementing the scheme efficiently will prove challenging.

9. CONCLUSIONS

In this paper, an overview of DASWAM, an implementation scheme for DDAS, which allows the exploitation of dependent and-parallelism in full Prolog, was presented. Results from DASWAM system (mainly the simulator, supported by the initial results from the parallel implementation), including two application programs, were also presented. The results suggest that DDAS/DASWAM is able to extract significant parallelism from at least some existing Prolog applications without too much effort. They also suggest that the overhead for doing so is not prohibitively expensive, in both memory usage and execution speed. The results also suggest that the design decision to allow the user to provide annotations to indicate where parallelism should be exploited is justified: this serves both to extract parallelism from situations in which an automatic parallelization scheme would fail, as in Orsim and, it also allows the user the option of not exploiting parallelism in situations where the cost of doing so is greater than the benefit, as in the case of the "DAP1" versus "DAP2" annotations for the wms1 program.

The results also suggest that by being able to exploit both deterministic and nondeterministic DAP, it may be easier to extract parallelism from existing Prolog programs than either independent and-parallelism or deterministic and-parallelism.

Much work remains to be done. This includes:

- Continue the development/optimization of the DASWAM implementation.
- Use of compile-time analysis to automatically generate some of the ECGE annotations.

[25]The difference in execution time is measured to be 24% on SICStus 2.1. This is slightly less than the differences in Table 1, probably because of garbage collection.

- Refinement and optimization of the memory management. This includes implementing a garbage collector.
- Adding exploitation of or-parallelism to the model, as is proposed in the Prometheus model [27].
- Extending the language to include features such as constraints.
- Further study of the system using more application programs.

I am very grateful to all the people who helped with the research presented in this paper, and in particular Mats Carlsson for his help with SICStus and understanding the SICStus WAM, Manuel Hermenegildo for enlightening discussions on the implementation of &-Prolog and general encouragement, and Vítor Santos Costa for his help in understanding the WAM and many helpful discussions on many aspects of DDAS and DASWAM. I also thank Rong Yang and Dewi Munaf for their help with the wms program. I also thank SICS for the use of their parallel processors, especially the Symmetry. Part of this work was partially supported by an ORS grant while I was a student, and subsequently I was supported by a BT (British Telecom) Fellowship for a year at the University of Bristol. Finally, I would like to thank the anonymous referees for their comments on earlier versions of this paper.

REFERENCES

1. Ali, K. A. M. and Karlsson, R., Full Prolog and scheduling or-parallelism in Muse, *Int. J. Parallel Program.* 19(6):445–475 (1990).
2. Baron, U. C., Chassin de Kergommeaux, J., Hailperin, M., Ratcliffe, M., Robert, P., Syre, J.-C., and Westphal, H., The parallel ECRC Prolog system PEPSys: An overview and evaluation results, in: *Proc. Int. Conf. on Fifth Generation Computer Systems 1988*, Vol. 3, 1988, pp. 841–850.
3. Carlsson, M., *SICStus Prolog Internals Manual*, Swedish Institute of Computer Science, Box 1263, S-163 12 Spånga, Sweden, Jan. 1989.
4. Chikayama, T., Fujise, T., and Sekita, D., A portable and efficient implementation of KL1, in: T. Chikayama and E. Tick (eds.), *Workshop on Parallel Logic Programming and its Programming Environments*, 1994.
5. Codognet, C. and Codognet, P., Non-deterministic stream and-parallelism based on intelligent backtracking, in: G. Levi and M. Martelli (eds.), *Logic Programming: Proc. Sixth Int. Conf.*, MIT Press, 1989, pp. 63–79.
6. Crammond, J., Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors, Ph.D. thesis, Heriot-Watt University, 1988.
7. DeGroot, D., Restricted and-parallelism, in: *Proc. Int. Conf. on Fifth Generation Computer Systems*, 1984, pp. 471–478.
8. Drakos, N., Unrestricted and-parallel execution of logic programs with dependency directed backtracking, in: N. S. Sridharan (ed.), *Proc. Eleventh Int. Joint Conf. on Artificial Intelligence*, Aug. 1989, Vol. 1, pp. 157–162.
9. Gregory, S., Design, Application and Implementation of a Parallel Logic Language, Ph.D. thesis, Imperial College of Science and Technology, University of London, 1985.
10. Gupta, G., Hermenegildo, M. V., Pontelli, E., and Santos Costa, V., ACE: And/or-parallel copying-based execution of logic programs, in: *Logic Programming: Proc. Eleventh Conf.*, MIT Press, June 1994, pp. 93–110.
11. Hermenegildo, M. V., An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel, Ph.D. thesis, University of Texas, Austin, 1986.
12. Hermenegildo, M. V. and Greene, K., The &-Prolog system: Exploiting independent and-parallelism, *New Gen. Comput.* 9(3,4):233–257 (1991).

13. Hermenegildo, M. V. and Rossi, F., On the correctness and efficiency of independent and-parallelism in logic programs, in: *1989 North American Conf. on Logic Programming*, MIT Press, Oct. 1989.
14. Kowalski, R. A., Algorithm = logic + control, *Commun. ACM* 22(7):425–436 (1979).
15. Lusk, E. L., Butler, R., Disz, T., Olson, R., Overbeek, R. A., Stevens, R., Warren, D. H. D., Calderwood, A., Szeredi, P., Haridi, S., Brand, P., Carlsson, M., Ciepielewski, A. and Hausman, B., The Aurora or-parallel Prolog system, *New Gen. Comput.* 7(2,3) (1990).
16. Munaf, D., Workforce management in Andorra-I, Technical Report MAIN-WP 1203. Issue 1, British Telecom Research Laboratory, 1993.
17. Muthukumar, K. and Hermenegildo, M. V., Efficient methods for supporting side effects in independent and-parallelism and their backtracking semantics, in: G. Levi and M. Martelli (eds.), *Logic Programming: Proc. Sixth Int. Conf.*, MIT Press, June 1989, pp. 80–97.
18. Naish, L., *Negation and control in prolog, Lecture Notes in Computer Science*, Vol. 238, Springer-Verlag, 1986.
19. Naish, L., Parallelizing NU-Prolog, in: *Fifth Int. Conf. and Symp. on Logic Programming*, University of Washington, MIT Press, August 1988, pp. 1546–1564.
20. O'Keefe, R. A., *The Craft of Prolog*, MIT Press, 1990.
21. Olthof, I. W., An Optimistic AND-Parallel Prolog Implementation, Master's thesis, Department of Computer Science, University of Calgary, 1991.
22. Santos Costa, V., Warren, D. H. D., and Yang, R., The Andorra-I engine: A parallel implementation of the basic Andorra model, in: *Logic Programming: Proc. Eighth Int. Conf.*, 1991, pp. 825–839.
23. Santos Costa, V., Warren, D. H. D., and Yang, R., The Andorra-I preprocessor: Supporting full Prolog on the basic Andorra model, in: *Logic Programming: Proc. Eighth Int. Conf.*, 1991, pp. 443–456.
24. Shapiro, E., Concurrent Prolog: A progress report, *Computer* 19(8) (1986).
25. Shen, K., An Investigation of the Argonne Model of Or-Parallel Prolog, Master's thesis, University of Manchester, 1986. Available as University of Manchester Computer Science Technical Report UMCS-87-1-1.
26. Shen, K., Exploiting and-parallelism in Prolog: The dynamic dependent and-parallel scheme (DDAS), in: *Logic Programming: Proc. Joint Int. Conf. and Symp. on Logic Programming*, MIT Press, 1992, pp. 717–731.
27. Shen, K., Studies of And/Or Parallelism in Prolog, Ph.D. thesis, Computer Laboratory, University of Cambridge, 1992.
28. Shen, K., Implementing dynamic dependent and-parallelism, in: *Logic Programming: Proc. Tenth Int. Conf.*, MIT Press, 1993, pp. 167–183.
29. Shen, K., Improving the execution of the dependent and-parallel Prolog DDAS, in: C Halatis, D. Maritsas, G. Philokyprou, and S. Theodoridis (eds.), *PARLE'94 Parallel Architectures and Languages Europe*, Springer-Verlag, 1994, pp. 438–452. Published as Lecture Notes in Computer Science 817.
30. Shen, K. and Hermenegildo, M. V., A simulation study of or- and independent and-parallelism, in: V. Saraswat and K. Ueda (eds.), *Logic Programming: Proc. 1991 Int. Symp.*, MIT Press, 1991, pp. 135–151.
31. Shen, K. and Warren, D. H. D., A simulation study of the Argonne model for or-parallel execution of Prolog, in: *Proc. Fourth Symp. on Logic Programming*, IEEE Computer Society Press, Sept. 1987.
32. Szeredi, P., Performance analysis of the Aurora or-parallel system, in: E. L. Lusk and R. A. Overbeek (eds.), *Logic Programming: Proc. North American Conf., 1989*, Vol. 2, MIT Press, Oct. 1989, pp. 713–732.

33. Tebra, H., Optimistic And-Parallelism in Prolog, Ph.D. thesis, Vrije Universiteit te Amsterdam, 1989.
34. Tick, E., Memory performance of Lisp and Prolog programs, in: E. Shapiro (ed.), *Third Int. Conf. on Logic Programming*, Springer-Verlag, 1986, pp. 642–649. Published as Lecture Notes in Computer Science 225.
35. Ueda, K., Guarded Horn Clauses, Ph.D. thesis, University of Tokyo, 1986.
36. Ullman, J. D., *Principles of Database and Knowledge-Base Systems*, Vol. II, Computer Science Press, Rockville, MD, 1989.
37. Warren, D. H. D., An Abstract Prolog Instruction Set, Technical Note 309, SRI International, Menlo Park, CA, 1983.
38. Warren, D. S., Efficient Prolog management for flexible control strategies, in: *Proc. Second Symp. on Logic Programming*, 1984.